

NAVAL POSTGRADUATE SCHOOL

Monterey, California

AD-A205 742



THESIS

MANAGING SOUND IN A RELATIONAL MULTIMEDIA DATABASE SYSTEM

by

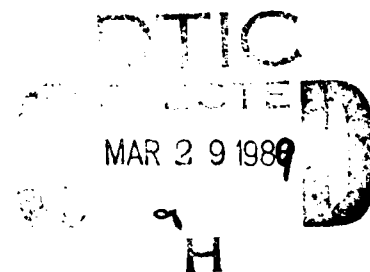
Gregory Russell Sawyer

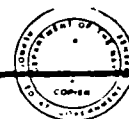
December 1988

Thesis Advisor:

Vincent Y. Lum

Approved for public release; distribution is unlimited.





REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b RESTRICTIVE MARKINGS	
2a SECURITY CLASSIFICATION AUTHORITY		3 DISTRIBUTION AVAILABILITY OF REPORT Approved for public release; Distribution is unlimited	
2b DECLASSIFICATION/DOWNGRADING SCHEDULE		5 MONITORING ORGANIZATION REPORT NUMBER(S)	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)		5 MONITORING ORGANIZATION REPORT NUMBER(S)	
6a NAME OF PERFORMING ORGANIZATION NAVAL POSTGRADUATE SCHOOL	6b OFFICE SYMBOL (If applicable) CODE 52	7a NAME OF MONITORING ORGANIZATION NAVAL POSTGRADUATE SCHOOL	
6c ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000		7b ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	
8a NAME OF FUNDING SPONSORING ORGANIZATION	8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c ADDRESS (City, State, and ZIP Code)		10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. PROJECT NO. TASK NO. WORK UNIT ACCESSION NO.	
11 TITLE (Include Security Classification) MANAGING SOUND IN A RELATIONAL MULTIMEDIA DATABASE SYSTEM			
12 PERSONAL AUTHOR(S) SAWYER, GREGORY R.			
13a TYPE OF REPORT MASTER'S THESIS	13b TIME COVERED FROM TO	14 DATE OF REPORT (Year, Month, Day) 1988 December	15 PAGE COUNT 108
16 SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
17 COSATI CODES FIELD GROUP SUB-GROUP		18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number) multimedia, multimedia database, digitizing, encoding, abstract data type, sound, operations, sound handling	
19 ABSTRACT (Continue on reverse if necessary and identify by block number) Sound, in all of its varied forms, is an important and widely used medium for the transmission of information. The widespread use of computers has greatly increased the breadth and depth of our information processing abilities. Yet the limited sensory functionality of computers has traditionally dictated a predominantly alphanumeric or "textual" communications interface standard. This thesis concentrates on the effective manipulation (i.e., capture, storage and retrieval) of sound data in a relational database system. It introduces the concept of an abstract data type of type SOUND which permits			
20 DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USES		21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a NAME OF RESPONSIBLE INDIVIDUAL Professor Vincent Y. Lum		22b TELEPHONE (Include Area Code) (408) 646-2449	22c OFFICE SYMBOL Code 52Lu

BLOCK 19(continued)

a level of sophistication in data manipulations that is beyond the capabilities of current systems. Such sophistication is accomplished through the use of a set of data manipulation operations which effectively hide the representation of the SOUND data structure from the user. As a result, the current familiarity of the user's view of the database remains unchanged when extended to the multimedia information processing environment.



Accession For

NTIS GRA&I ☒DTIC TAB ☐Unannounced ☐

Justification on

Distribution/

Availability Codes

A-1

Approved for public release; distribution is unlimited.

**MANAGING SOUND IN A RELATIONAL MULTIMEDIA
DATABASE SYSTEM**

by

Gregory R. Sawyer
Lieutenant Commander, United States Navy
B.S., United States Naval Academy, 1977

Submitted in partial fulfillment of the
requirements for the degree of

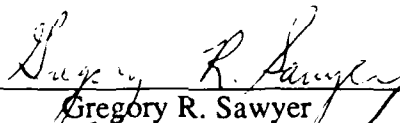
MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

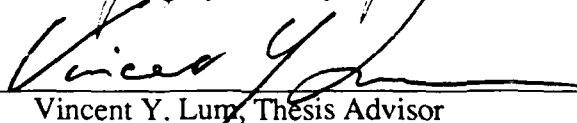
NAVAL POSTGRADUATE SCHOOL

December 1988

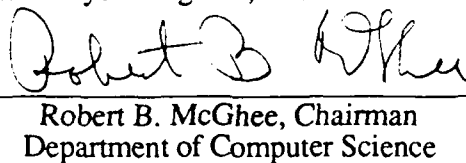
Author:

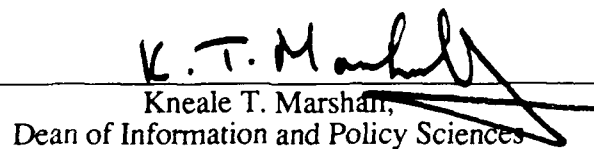

Gregory R. Sawyer

Approved by:


Vincent Y. Lum, Thesis Advisor


Klaus Meyer-Wegener, Thesis Co-Advisor


Robert B. McGhee, Chairman
Department of Computer Science


Kneale T. Marshall,
Dean of Information and Policy Sciences

ABSTRACT

Sound, in all of its varied forms, is an important and widely used medium for the transmission of information. The widespread use of computers has greatly increased the breadth and depth of our information processing abilities. Yet the limited sensory functionality of computers has traditionally dictated a predominantly alphanumeric or "textual" communications interface standard. This thesis concentrates on the effective manipulation (i.e., capture, storage and retrieval) of sound data in a relational database system. It introduces the concept of an abstract data type of type SOUND which permits a level of sophistication in data manipulations that is beyond the capabilities of current systems. Such sophistication is accomplished through the use of a set of data manipulation operations which effectively hide the representation of the SOUND data structure from the user. As a result, the current familiarity of the user's view of the database remains unchanged when extended to the multimedia information processing environment.

TABLE OF CONTENTS

I.	INTRODUCTION	1
	A. BACKGROUND	1
	B. PURPOSE OF THESIS	2
	C. AN OVERVIEW	3
	D. ORGANIZATION OF THESIS	6
II.	A DESCRIPTION OF SOUND ENCODING TECHNIQUES	7
	A. INTRODUCTION	7
	B. INTRODUCTION TO SOUND	7
	C. TYPES OF ENCODING	12
	1. Waveform Encoding	13
	2. Parameter Encoding	16
	D. TRANSFORMATION BETWEEN TYPES	18
III.	MULTIMEDIA MASS STORAGE DEVICES	20
	A. CONVENTIONAL SYSTEMS	20
	B. OPTICAL DISKS	22
IV.	OVERVIEW OF RELATED WORK	25
	A. INTRODUCTION	25
	B. OVERVIEW	25
V.	THE SOUND DATABASE AND INFORMATION SYSTEM	30
	A. SOUND MEDIA MANAGEMENT	30
	B. A MULTIMEDIA ARCHITECTURE	31
	C. THE SOUND DATA TYPE	37
	1. Sound Data Organization	37
	2. The User's View	39

3. Implementation of the Abstract Data Type	46
VI. DESCRIPTION OF A SOUND MANAGEMENT PROTOTYPE	51
A. ARCHITECTURE OF A PROTOTYPE	51
B. MODEL SPECIFIC EQUIPMENT	53
C. IMPLEMENTATION CONSIDERATIONS OF THE MODEL	55
VII. SUMMARY AND CONCLUSIONS	57
A. REVIEW OF THESIS	57
B. APPLICATIONS	57
C. FUTURE RESEARCH AREAS	59
LIST OF REFERENCES	60
APPENDIX A - THE SQL PREPROCESSOR OVERVIEW	62
APPENDIX B - THE INTERNAL SOUND HANDLER FUNCTIONS	68
BIBLIOGRAPHY	96
INITIAL DISTRIBUTION LIST	99

LIST OF FIGURES

Figure 1. Properties of a simple sinusoidal waveform	9
Figure 2. DSP Hardware Configuration for Coder Designs	13
Figure 3. How an Optical Disk Works	23
Figure 4. The Conceptual Model of a Multimedia Object	34
Figure 5. The Components of a Multimedia System	36
Figure 6. Architecture of the Sound Media Prototype	52

I. INTRODUCTION

A. BACKGROUND

Recent advances in both hardware and database applications have furthered the idea of achieving the representational storage of real world information objects within the computer system. Applications must be developed to effectively and efficiently handle the vast diversity of this real world multimedia data. In addition to text, graphics, images, and sound are gaining greater importance and must also be effectively integrated. It is this ability to access more than mere textual information that embodies the principal driving force behind the development of multimedia information systems.

Sound, in all of its varied forms, is an important and widely used medium for the transmission of information. The widespread use of computers has greatly increased the breadth and depth of our information processing abilities. Yet the limited sensory functionality of computers has traditionally dictated a predominantly alphanumeric or "textual" communications interface standard.

The efficient storage and retrieval of multimedia information has recently sparked numerous research efforts. Several prototypes, primarily within the area of office automation, have been developed. Many of these systems will be discussed in more detail in Chapter IV. The ultimate use of multimedia information systems lies in the creation of an artificial reality which is totally controlled by the user. There are some obvious benefits to this approach. By involving more sensory stimuli, better data correlations and increased information accesses are achievable.

Multimedia information can be of special interest to the Department of Defense. Voice, for example, is much easier to encrypt digitally rather than through the use of an analog process. The superior quality of digital encryption is a definite advantage. Additionally,

the ability to access computer systems through the use of the telephone provides far reaching applications in the area of sound management.

Each type of information medium requires its own unique modes of handling. But the fundamental difficulty faced by users of multimedia database systems lies in handling the rich semantics of the multimedia data. Unfortunately, the ability to properly manage unconventional data has proven an extremely difficult and highly complex task. Better ways must be found which reduce the complexity involved in handling multimedia data.

B. PURPOSE OF THESIS

The underlying focus of this thesis lies in the research of effective means by which sound can be managed (i.e., captured, stored, retrieved, edited) in a multimedia information and database system. This research is designed to answer the following questions:

1. What is the feasibility of developing audio (sound) storage and retrieval capabilities using only conventional programming tools in conjunction with existing off-the-shelf technologies?
2. What is an appropriate design for a data base which permits the querying of unstructured data in the form of sound?
3. What characteristics of sound must be captured and stored which would aid in the realization of such a design?
4. What functions are required to allow users to manipulate sound data?
5. What kind of queries of sound data is meaningful and how can this criteria be achieved?
6. How can the development of this technology be used to meet the growing real time requirements of today's highly technical Navy?

Once a discussion of sound data's capture, storage, retrieval and implementation details has been presented, an architecture of a prototype will be proposed. This phase will explore the feasibility of integrating existing hardware and software components into a functional multimedia information system which incorporates the management of sound

data. The utilization of "off-the-shelf" technology will serve as the underlying criteria for the design of this prototype.

C. AN OVERVIEW

The management of sound data in a database management system offers several unique challenges that often go unnoticed or do not apply in other media forms. For example, the *real time* aspect of acoustic energy must be captured and expressed if a displayed (i.e., played) sound is to have relevance and meaning. That is, a playback that is too fast or too slow may be entirely unintelligible. When converting a sound into a digital representation which can be managed by the computer system, a fixed sampling rate must be employed. Otherwise, a substantial loss of quality and information will result.

Information retrieval in conventional (i.e., alphanumeric data to be called textual) database systems has enjoyed considerable success over the past two decades. However, extending these accomplishments to other media has not met with similar success. The management of sound, unfortunately, has also proven to be no exception. Unlike text based systems, sound features cannot be readily extracted from a data file without a complex sequence of steps, many of which remain to be defined. Additionally, the enormous data resulting from an acceptable sampling rate of an acoustic input is--by present system standards--far too large to be reasonably stored within the database itself. Future DBMS, however, are expected to handle such voluminous data storage requirements as part of their standard operations.

Current technology is limited in its ability to handle sound data storage to an equal extent as that afforded the more familiar textual form. This is not meant to imply that such limitations will continue to be the state of future systems. Many of the sound handling systems reviewed in the literature have centered their research and development efforts in the area of office automation. The use of sound data for inner office taskings such as audio

memos and annotations constitute the restricted integration of sound within the scope of current multimedia information systems.

A few prototypes under development, however, do offer the promise of increased sound data manipulation techniques and a wider variety of applications. In addition to the basic sound manipulation features of record, store and play, such systems as the Etherphone and Diamond (see Chapter IV) also offer the ability to edit and link sound data segments. It is from this perspective that the research direction for this thesis has evolved. Through the introduction of the concept of an abstract data type of type SOUND, we are able to achieve a level of sophistication in data manipulations that is beyond the capabilities of current systems. By providing a set of operations which can be used to manipulate the SOUND data structure, we effectively hide the representation of the data structure from the user. Through this approach, the current familiarity of the user's view of the database need never change.

One of the major drawbacks faced in handling digitized acoustic data is the large data volume which results. These problems have been under study for many years by telephone companies concerned with minimizing the amount of information being transmitted without a corresponding loss in signal quality. Since the Nyquist theorem states that sampling rates of twice the highest frequency present are sufficient to capture all of an input sound, megabyte volumes for a few minutes of captured sound is quite the norm. Understandably, the efficient management of large amounts of data in a database is always tricky, regardless of what the actual data may represent.

To answer queries with regard to sound, a person engages well established mental capacities to analyze, synthesize and interpret the information. More useful information, however, is often obtainable from the context of the sound. This includes emphasis such as the vocal intonations and inflections associated with speech. The ability to extract certain

features from the sound data is, in itself, a wide open area for research. Realistically, however, the user of the multimedia database system should not expect this level of information extraction capability from current technology.

The approach proposed by Meyer-Wegener, et al. [Ref. 1], is to abstract the contents of sound data, image data and other forms into words or text. By storing the textual description of the media in the database, searching on the basis of data content is now possible. This description is manually entered by a human user. Such information extrapolation methods will always result in some loss of information. For querying purposes, however, this loss is acceptable. Moreover, current technology does not allow us to go beyond this level of sophistication when querying of sound data content is desired.

The architecture presented in Chapter VI uses this approach. Each medium will be represented by three parts: raw data, registration data and description data. Raw data is a bit string. For sound, this will be the linearly stored digitized samples. Registration data is the data which enables the proper decoding of the raw data for the device on which it will be displayed. Description data relates to the "semantics" or contents of the raw data and will be entered into the database by the user.

One of the problems associated with multimedia databases is portability. For example, although sound data files may be transferred between different host computers, the information may be totally unusable unless the new host is aware of the encoding algorithm used on the original sound. Without the ability to properly decode a sound data file, properly accessing the data becomes a serious problem. The nature of these problems will be further discussed in Chapter II.

D. ORGANIZATION OF THESIS

Chapter II discusses the nature of sound and describes several sound encoding techniques. Also discussed are the problems relating to translations between different encoding algorithms.

Chapter III describes the operation of optical storage devices and their impact on database management. Effective mass storage devices are an essential part of multimedia database management systems.

Chapter IV discusses other related work with respect to multimedia database systems. Of primary interest is the handling of acoustic data within the database.

Chapter V provides an overview of the relational data model designed for the prototype. A discussion of the SOUND data type is presented. Also, several handling techniques with regard to data searches will be discussed.

Chapter VI describes the components of the prototype and the architecture upon which it was designed. The specific equipment used in the development of this prototype is presented. The functions required for the implementation of the model are also discussed.

Chapter VII presents conclusions drawn from the research and implementation of this thesis, plus a projection of the applications for which this prototype may be used. Appendix A provides a brief overview of the SQL preprocessor functions as they relate to sound. Appendix B lists the various internal "C" functions that provide the interface to the software driver functions of the equipment used in the development of the model.

II. A DESCRIPTION OF SOUND ENCODING TECHNIQUES

A. INTRODUCTION

In this chapter, some of the basic properties of sound are discussed. Many of the definitions pertaining to sound are also presented. These form the basis of our discussions in the following chapters. With that background established, a description of several sound encoding techniques can be discussed. This backdrop will enable a viable, though limited, discussion to ensue regarding the transformation of data between different data encoding schemes.

B. INTRODUCTION TO SOUND

Sound is created by vibrations from some source. Vibrations can be transmitted through various media. Our primary concern will center around those transmitted through the air to the ear.

By vibrating air molecules to cause compression and rarefaction, acoustic energy is transmitted. Speech is a good example of acoustic energy. The motion of the vocal cords caused by air rushing past them sets the surrounding air molecules to vibrating. This vibration causes other adjacent molecules to vibrate. The motion is carried through the air to the outer ear where it is collected and focused into the inner ear through the auditory canal, causing the eardrum to vibrate. This vibration is sensed as *sound*.

Two primary attributes of sound are *frequency* and *amplitude* (or intensity). A sound generating source such as a tuning fork can be used to help visualize *frequency*. When a tuning fork is struck, it moves back and forth at a fixed rate. This alternation creates corresponding increases and decreases in the air pressure. The compressed and rarefied air pressure trail traveling away from the tuning fork follows a sinusoidal (or sine) function.

This trail, when plotted on the X-Y axes in which compression or its amplitude is represented by the X-axis and time by the Y-axis, forms a *waveform*.

One feature of the simple sine wave is that the shape of the waveform above its midline is the same as the shape of the waveform below the midline. The distance between any two successive peaks (i.e., successive compressions) in a waveform is called the *period*. The *period* is measured in seconds. *Frequency* is the number of peaks (or cycles) that occur in a second. These cycles per second are known as Hertz (Hz). Frequency is equal to the inverse of the period. The human ear is sensitive to frequencies in the range of 20 to 20,000 Hz. The height of the wave is called the *amplitude*. Amplitude indicates the relative loudness of a sound.[Ref. 2] Some of the basic sound related properties of the simple sine wave are depicted in Figure 1.

Most sounds have complex waveforms. Complex waves are composed of many frequencies with various amplitudes superimposed (i.e., added together) on top of one another. The shape of the wave lends tonal qualities. In general, the smoother the wave, the cleaner, clearer and sweeter the sound. A square wave, for example, would sound harsh as compared with the steady resonance of a sinusoidal wave.

Since complex waveforms contain multiple frequencies, it is sometimes easier to visualize these frequencies as members of a group or range of frequencies. By grouping a set of frequencies together we form a frequency subset, referred to as a *band*. The range of the frequencies in the band is called a *bandwidth*. For example, frequencies within a range of 15,000 to 20,000 Hz could be considered a *high band*, while those occurring within a range of 20 to 400 Hz would constitute a relatively *low band*. A *pass filter* (or just *filter*) is designed to restrict or limit the relative bandwidths of frequencies that pass through it. A low pass filter, for example, is designed to permit only low frequencies to pass.

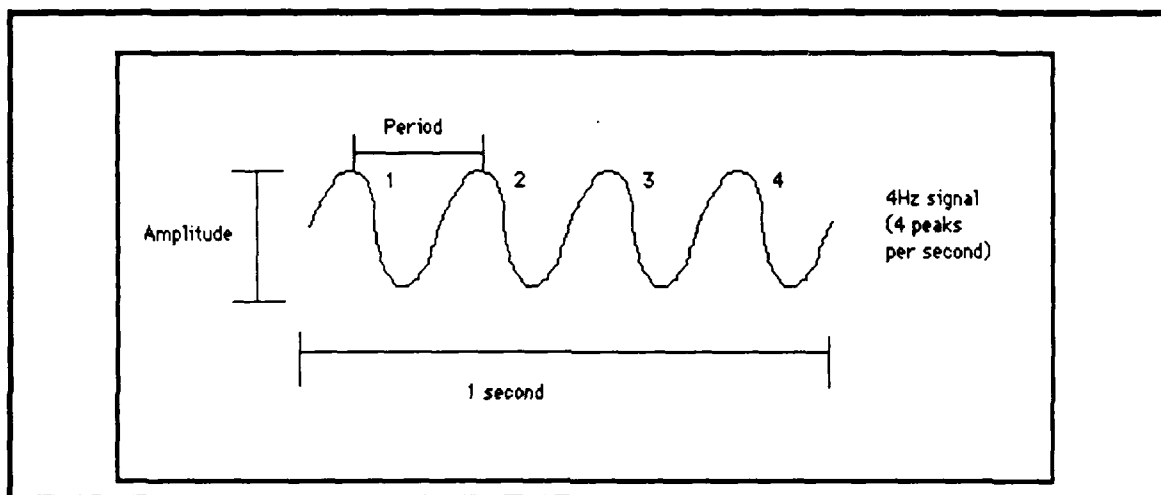


Figure 1. Properties of a simple sinusoidal waveform

One of the most common methods used to store sound in a computer is to convert the analog sound signal into a digital representation through a technique known as *sampling*. Samples of the amplitude of the waveform are taken at evenly spaced intervals of time using an analog-to-digital converter (ADC). The process of converting a sample of the waveform's energy level into a numerical quantity is known as *digitizing*. Thus each sample captures the relative volume (i.e., contribution of multiple frequencies) and energy level (amplitudinal measurement) of a sound signal at a certain point in time. The original waveform can be reconstructed from the samples, provided the sampling rate was sufficiently high enough. By increasing the rate of sampling, more of the higher frequencies can be captured. The *fidelity or quality* of a digitized sound is a reflection of the bandwidth or range of frequencies that has been captured. A sound consisting of frequencies of up to 20,000 Hz when captured at 20,000 samples per second would have a higher *quality* than the same sound captured at 10,000 samples per second.

Human speech is generally within the range of 500 to 5000 Hz. At the upper end of this range, the Nyquist theorem says that a sampling rate of 10,000 Hz would be sufficient to capture all of the information within this 5KHz bandwidth. More formally, Nyquist proved that

... if an arbitrary signal has been run through a low-pass filter of bandwidth H , the filtered signal can be completely reconstructed by making only $2 \cdot H$ (exact) samples per second. Sampling the line faster than $2 \cdot H$ times per second is pointless because the higher-frequency components that such samplings could recover have already been filtered out. [Ref. 3]

The chart of Table 1 summarizes the effects of sampling at different sampling rates.

Downsampling refers to sampling rates that are less than twice the highest filtered frequency present in a signal.

TABLE 1. THE EFFECTS OF DOWNSAMPLING

Sampling Rate	Frequency Range Recorded	Memory or Disk Bytes Used per Second	Max length of Sound per Mbyte of Ram or Disk
22,000	0 - 10KHz	22K	45 seconds
16,000	0 - 8KHz	16K	62.5 seconds
11,000	0 - 5KHz	11K	90 seconds
8,000	0 - 4KHz	8K	125 seconds
5,500	0 - 2.5KHz	5.5K	3 minutes

When a sound is digitized, the value of the amplitude is restricted to a range specified by the number of bits used to *digitize* the sample. For example, if eight bits per sample were used, the range would be from 0 to 255; if 16 bits per sample were used, the range

would be from 0 to 65535. Each sample is rounded off to the nearest integer. This process is known as *quantization*. If the amplitude of a wave is greater than the upper range, the "top" and "bottom" of the wave are cut off in order for the wave to "fit" within this range. This effect is known as *clipping*. Clipping causes distortion of the sound since it tends to produce sharp corners on waveforms. This results in sound signals that sound harsh. *Dynamic range* is a decibel (dB) measure of the difference between the loudest sound that can be recorded (without clipping) and the softest sound. The dynamic range of the human ear is around 120 dB.[Ref. 2]

An aspect of sound quality is denoted by the number of bits per sample used in the storage of digitized sound data. One method for reducing the amount of storage required for sound data is through data *compression* algorithms. A compression algorithm described in [Ref. 2], stores only one bit per sample instead of the entire eight bits or more resulting from the analog-to-digital conversion process. Viewing this method merely as an example, we could expect the resulting recorded data to be virtually useless upon playback. More specifically, *compression* algorithms reduce the average number of bits per sample of the resulting sound data values.

The primary disadvantage of data compression is the general loss of discriminability between syllables, words and sometimes even phrases that occur when the captured sound is reproduced. By taking advantage of the redundancy which occurs between successive samples of limited bandwidth waveforms, this loss can be perceptively reduced. There are encoding algorithms in use which can reduce the data storage requirement without a corresponding loss in sound quality. A few of these techniques are briefly discussed in the next section.

C. TYPES OF ENCODING

There are several methods commonly used to encode voiced sounds. Many of these techniques were developed in the field of telephone communications and have found their way into other venues of speech technology and the general field of sound capture and storage. The use of software in the encoding process has been completely and effectively replaced by the growing technological advancements in hardware.

Once the sound has been digitized, the next step is to try and reduce the number of bits needed through an assortment of statistical techniques, some of which are discussed below. All compression methods are based upon the principle that the sound signal changes relatively slowly compared to the sampling frequency. This means that much of the information at the digital level is redundant.

Through the use of special purpose signal processors and microprocessors, a variety of bandwidth reduction methods have been demonstrated. The analog signal is presented to a μ -Law codec (coder-decoder) chip where it is filtered and digitized, and then fed to a digital signal processor (DSP) chip where it is encoded. Figure 2 reprinted from [Ref. 4] presents an overview of the operation of a low-to-medium complexity DSP coder design. The μ -Law codec is a pulse code modulation chip (see next section) which performs the actual analog-to-digital and digital-to-analog (A/D and D/A) signal conversions. The DSP, a special purpose signal processor, is a powerful, single-chip, programmable microprocessor which performs the actual algorithmic encoding (statistical methods involved) of the digital PCM-signal.

All encoding techniques are realized by filtering and digitizing the analog signal, analyzing short segments of it, then encoding prior to transmission or storage. *Waveform encoding* and *parameter encoding* are the two broad categories we will use to summarize these schemes. The definitions presented below are encapsulations of information found in [Refs. 4, 5, and 6]. The actual mathematical formulas embodied in the discussions of the

following technique are not necessary to the scope of this thesis and therefore will not be expounded upon.

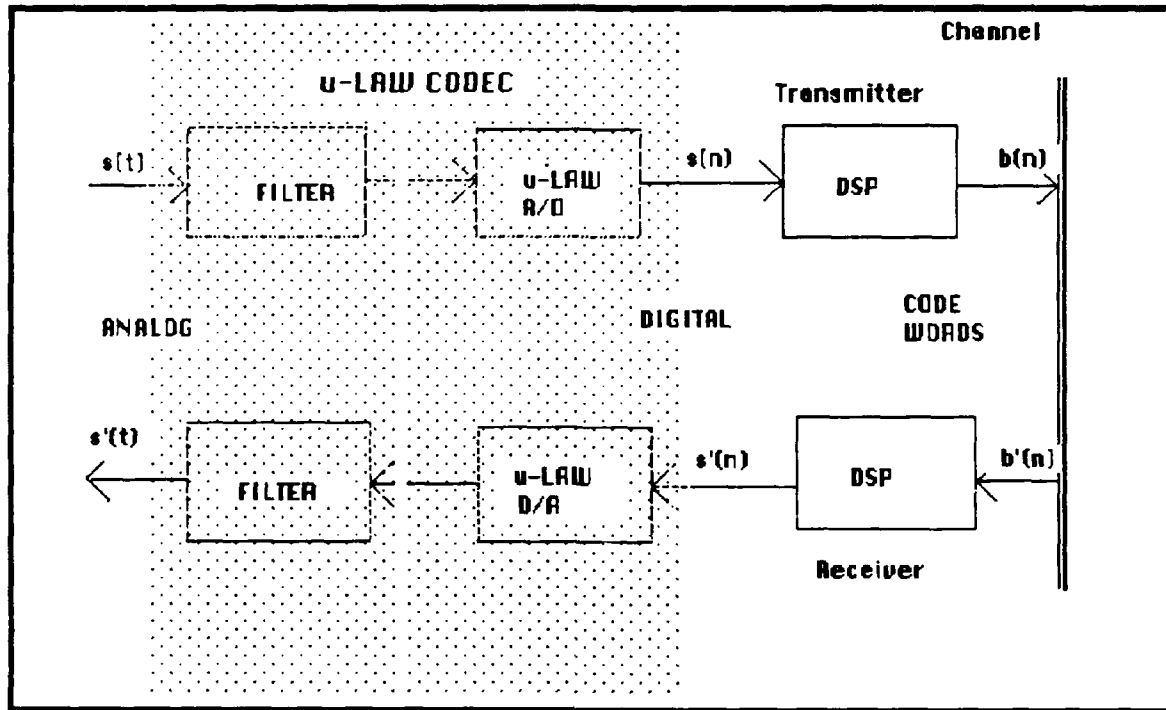


Figure 2. DSP Hardware Configuration for Coder Designs

1. Waveform Encoding

One of the most direct forms of waveform encoding is *pulse code modulation* (PCM) [Ref. 5]. It is the basic foundation of all waveform encoding schemes in use today. The amplitude of the sound is sampled at a fixed rate (typically 8000 bits per second for speech) and converted into digital information using an A/D converter. Each amplitude measurement (or sample) may require from 6 - 16 bits, depending on the PCM output of the codec in use.

Differential pulse code modulation (DPCM) consists of not outputting the digitized amplitude, but rather the difference between the current value and the previous one. Since amplitude differences of 32 or more on a normalized scale of 0 to 255 are unlikely, five bits should suffice instead of eight. If the signal does jump wildly, the encoding logic may require several sampling periods to "catch up." The error introduced for speech can generally be ignored. [Ref. 5]

Delta modulation (DM), a variation of the compaction method, requires each sample to differ from its predecessor by a minimum amount, either +1 or -1. A single bit is generated, telling whether the new sample is above or below the previous one. Delta modulation differs from the compaction algorithm in that alternate samples are compared to determine the relative energy level of the new sample vice reducing a digitized sample from eight or more bits down to only one. [Ref. 5] This approach is obviously unsatisfactory for rapidly changing signals since small level changes are assumed between samples. This is true even when each value represents or indicates different absolute amounts. For example, +1 may mean adding five to the previous amplitude. Delta modulation can also be implemented by encoding the slope of the changing waveform as one of several fixed values (other than +1 or -1) [Ref. 4]. These values can be permanently stored in the "on-chip" ROM of the DSP chip.

An improvement to differential PCM is achieved by taking a few of the previous samples and extrapolating (i.e., predicting) what the next value will be. Once the actual next value is obtained, the step size--the difference between the actual and the predicted signal--is adaptively quantized (or encoded). This method is known as *adaptive differential pulse code modulation* (ADPCM). [Ref. 4]

ADPCM is a low complexity technique and was one of the first encoder algorithms realized on the DSP. The discussions of [Ref. 4] show that the design is based on a

backward adaptive step-size algorithm with a fixed first-order predictor (possibly a "slope" computation) and a robust adapting step size. A predictor signal is generated by scaling the previously decoded signal by the predictor coefficient. This value is then subtracted from the input signal to form the difference signal. A table lookup is performed, based on the difference signal, to locate the quantization value. This table-based conversion process is known as adaptive quantization. By storing the step sizes and inverse step sizes in on-chip tables, the need for a divide in adaptively scaling the difference signal before and after quantization is avoided. This offers a tremendous advantage in real-time applications in terms of speed.

ADPCM offers tremendous flexibility in signal encoding and is in widespread use throughout the speech technology industry. This technique is used in the prototype development of the model described in Chapter VI. All data is packed 4 bit ADPCM. This means that at an 8 KHz sampling rate, only 4K bytes per second (32,000 bits/sec) are generated as opposed to 8K bytes per second (64,000 bits/sec) using standard PCM.

The next level of algorithmic complexity is that of *subband coding* (SBC). This approach has little advantage over ADPCM for 8 KHz sampled inputs. The two-band (there are many others) SBC scheme divides the input into two equally spaced high and low frequency bands with a filter bank. The two subband signals are then reduced in sampling rate and separately coded with ADPCM encoders. The reverse process takes place in the receiver (in the case of transmissions) and the digital-to-analog converter (DAC). Table 2 compares a few of the techniques mentioned above. [Refs. 4 and 6]

TABLE 2. DIGITAL ENCODING TECHNIQUES

PCM	87	92	98	111	122	131	135	129	...
DPCM	87	+5	+6	+13	+11	+9	+4	-6	...
DM (1)	87	+1	+1	+1	+1	+1	+1	-1	...
DM (2)	87	+3	+3	+4	+4	+4	+2	-3	...
ADPCM	87	(based on step sizes stored in tables)							

Other encoding techniques have been developed, but these typically entail a level of complexity that far exceeds our need of a basic understanding of the processes involved. The use of more bands coupled with *time domain harmonic scaling* (TDHS), for example, can greatly reduce the amount of bits per second of encoded data. [Ref. 4] explains that the TDHS algorithm compresses the input signal by a factor of two (or more) in bandwidth and sampling rate before passing it on to a SBC coder. The trade-off, however, is a tremendous increase in the level of complexity required to realize the encoding algorithm. The multiple DSP approach (i.e., DSPs connected in sequence) can also be used, but this leads to a multiprocessor system which requires precise communication and I/O synchronization between processors.

2. Parameter Encoding

Parameter encoders are typically referred to as *vocoders*. Vocoders have found widespread application in speech recognition and speech synthesis technology. [Ref. 5] explains how the spectral shape of the speech signal is encoded rather than the speech

waveform. The spectral shape denotes an instance of the frequency spectrum of a signal for a fixed period of time. A model of human speech production is used to obtain a compact representation of the speech signal. Bit rates as low as 400 bits/sec can be achieved, but the speech quality, at best, is only fair.

Signal characteristics are usually extracted in the frequency domain. (ie., certain frequency bands are used in the digitizing process). These are used to control a mathematical synthesis model to create an output speech signal whose waveform is perceived as similar to the original one. One example of this form of encoding is called *formant synthesis* which operates by decoding the spectral peaks (formants) of the signal.

Linear predictive coding (LPC) explained in [Ref. 5] is another example which can be interpreted as a simple model of the human vocal tract. It is basically a time domain operation which involves predicting the next sample of a waveform based on a linear combination of a set of spectral numbers of previous samples. LPC is common but is not as reliable as might be desired for general encoding. Nevertheless, LPC has demonstrated excellent applicability in speech generating devices such as talking consumer products. With LPC, encoding rates ranging between 1200 to 2400 bits per second (at a sampling rate of 8000 samples per second) have been achieved [Ref. 5].

Today, the speech coding technology is available to achieve a high speech quality at 16K bits per second (or 2K bytes per second at eight bits per byte). These techniques are often hybrids of the waveform and parameter encoding methods. We will not go into detail on their operation except to say that these techniques are more complex and far more costly than those used in standard PCM, ADPCM and LPC coders. Their advantage is that they can be implemented on a single very-large-scale-integration (VLSI) digital signal processor chip for real time encoding.

D. TRANSFORMATIONS BETWEEN TYPES

Once a sound signal has been successfully digitized, it can be stored or transmitted with considerable speed. However, for the receiving processor to properly use the received data, both the transmitter and the receiver must use the same encoding and decoding algorithms. Generally this is not a problem in a single workstation environment in which the ADC and the DAC conversions are both contained on the same chip. Nor is this a problem among different workstations, provided the same encoding algorithm is used in the sharing of data. This approach would serve equally well in the event that a central sound archive or file server were employed. The hardware implemented encoding and decoding algorithms are self-contained and perform all conversions automatically. Hardware implemented algorithms severely restrict the transformation of data between different encoding schemes.

The other compatibility aspect of data transformations involves the sampling rate used in the digitizing process. A 16 KHz sampling rate played back at the 8 KHz rate would produce a highly distorted output since the duration expected for each sample would be in error (16K-62.5 μ -sec/sample, 8K-125 μ -sec/sample). This duration is built into the chip and may be hardware selectable based on the desired rate of play. The sample size (number of bits per sample) must also be known to ensure proper boundaries are maintained between data samples.

The final consideration for achieving universal transformation between different encoding algorithms is through the use of an intermediate encoding scheme. Software drivers can be installed which will convert all data into PCM, the more basic form of the digitizing process. Other encoding techniques can be employed to convert the PCM data into more suitable algorithms for the chip in use. The driver, however, must be smart enough to recognize the encoding used on the incoming data before an effective transformation can occur. The necessity for data conversions in an algorithmically

heterogeneous encoding environment is a rather difficult problem which, unfortunately, cannot be avoided .

III. MULTIMEDIA MASS STORAGE DEVICES

The efficiency of multimedia storage requires the use of physical storage devices capable of handling massive data storage requirements. As a prelude to the discussion of specific types of storage devices, a reflection on the basic concepts of physical storage is offered. This brief synopsis is adapted from E. Bertino et al. [Ref. 7] in their discussions of query processing in a multimedia database.

To begin, physical storage is organized in *devices*, which can be either magnetic or optical. An example of a magnetic device is the familiar hard disk. Optical devices may consist of a single disk drive or be arranged in a *jukebox*, in which one device at a time is mounted. Devices are divided into *segments*, where a segment is a set of extents. Finally an *extent* is a physically contiguous region of secondary storage, such as a cylinder on magnetic storage or a set of sequential sectors on optical storage. Segments are used to store document and file indexes, text access structures, system tables and data instances. An optical system may be open or frozen, depending on whether or not writing is allowed.

A. CONVENTIONAL SYSTEMS

Multimedia database systems, typically referring to the integration of text, graphics, images and sound, have storage requirements that dwarf most conventional magnetic storage media. The minicomputer explosion of the 1980's has introduced a number of easily integratable secondary storage devices (hard disks) which have managed to alleviate a portion of the problem.

Since the highly common 5.25-inch magnetic floppy disk only holds a maximum of 800K bytes, the necessity for increased storage volume devices remains a crucial concern. The appearance of multi-megabyte secondary storage devices in the mid 1980's helped to

ease the problem, but was not an optimum solution. The capacity of magnetic hard disk storage devices has grown from a nominal 1M byte for microcomputers up to and exceeding 100+ Mbytes for large capacity (i.e., mainframe) systems. Additionally, the recent growth of local area networks and the use of network file servers has aided in the reduction of the data storage requirements.

But even this increased capacity has proven somewhat limited. Superficially, many of the compression algorithms discussed in Chapter II have managed to reduce the massive data storage requirements resulting from the digitization of sound. Unfortunately, the necessity for high fidelity reproductions or extended (i.e., long playing) recordings obviate most of the advantages gained. Maintaining a sufficient data storage medium for images and video presents an even more critical problem .

Consider the following example regarding this limitation. An uncompressed two minute sound recording sampled at 8K bytes/sec would require approximately 1Mbytes of storage. The use of standard 800 Kbyte 5.25-inch floppy disk is inherently unsuited for such enormous data storage requirements. Comparatively, a single image of 1024 x 1024 byte pixels would also require 1Mbyte of storage. When color bands are added, the image storage requirement becomes even larger. Similarly, a moving picture requires up to 24 frames (or images) per second in order to provide the sensation of continuous motion. The storage of uncompact digitized video would substantially overflow the boundaries of most magnetic media storage devices. Relatedly, the necessity for a high bandwidth transfer of data from the storage device to main memory and/or output devices could severely hamper the efficiency of video data management requirements. An interim solution to the mass storage dilemma has emerged in the form of the optical disk.

B. OPTICAL DISKS

The growing familiarity of optical disk technology is consummately linked to the music industry. The ability to store large amounts of digitized audio data with reproduction (i.e., playback) as clear, crisp and clean as the original was a welcomed relief to a highly disgruntled populace of audiophiles. Strangely enough, optical disk technology has quickly gained a foothold in office automation applications. Its continued use will have a significant impact on database management because it can incorporate data, text, image, video and audio information. This technology provides users with rapid access to a far greater amount of information on a single disk than ever before available on competing magnetic storage devices [Ref. 8].

Optical disks store information by means of microscopic pits on the disk. First, data is recorded on the disk by a laser beam, which burns the pits onto the media's reflective surface. The pits translate into binary code. The data is read by scanning the rapidly spinning rotating disk for differences in reflectivity due to the pits. Figure 3, reprinted from [Ref. 9], describes how an optical disk works.

Two types of optical disks have been introduced, with both gaining firm footholds within the computer industry. The first is the CD-ROM (Compact Disk - Read Only Memory), the version in which the data is permanently stored on the disk at the time the disk is pressed. The recorded information is protected by a layer of plastic. Thus, the disks are invulnerable to the damage that can be done to magnetic disks through head crashes or mishandling.

The second type of optical disk is the WORM (Write-Once-Read-Many) disk. Data may be recorded only once, but can be read back or reviewed many times. A single 4.25-inch CD-ROM (same size as used by the audio entertainment industry) is capable of storing up to 650 megabytes of data (of course the larger video disks can store more).

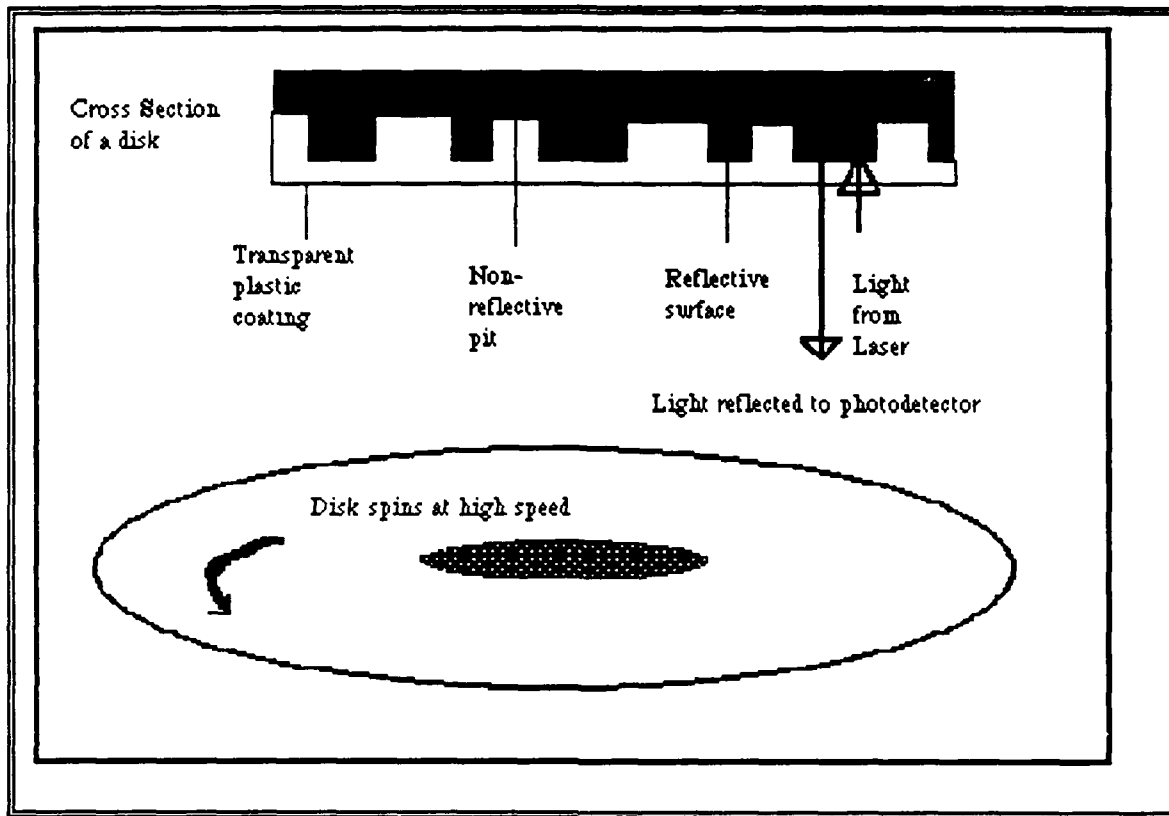


Figure 3. How An Optical Disk Works

That's as much data as can be stored on 1500 floppy disks (around 325,000 pages of text). WORM disk sizes have not been standardized and come in a variety of incompatible sizes (5.25-, 8-, 12- and 14-inch disks). A 12-inch WORM disk can store the equivalent of 400,000 pages of text. That equals 40 reels of magnetic tape or 10,000 frames of microfilm.

WORM disks are uniquely suited for use with multimedia databases. The storage capacity is large enough for most applications, including those involving graphics, sound and video presentations. Current WORM disk drives are SCSI adaptable to IBM/PC/AT and compatibles and could be included as an off-the-shelf peripheral storage device for the prototype presented in Chapter VII. Multimedia products will undoubtedly play a key role in optical disk applications of the near future.

Most current optical disk drives are dedicated to single, standalone PC's. Fortunately, however, the technology has been standardized to allow any drive to read any disk (of the same size). This makes disks and drives fully interchangeable. CD-ROM manufacturing companies, pressing plants, publishers and software developers have agreed on standards for both the hardware and software elements of CD-ROM. A significant milestone in the area of software standards is the 1988 decision by the International Standards Organization (ISO) to incorporate the *HIGH SIERRA* CD-ROM file format into a worldwide standard--ISO 9660. [Ref. 9]

Erasable (or reversible) optical disk technology is another area poised for future growth. This technology will combine the capacity and reliability of optical disks with the erasability and flexibility of magnetic media. The user can revise any data stored on them. Speculative drawbacks include a relatively high cost, high access time and currently limited application suitability. The limitations on random access would also need to be addressed and suitably resolved.

IV. OVERVIEW OF RELATED WORK

A. INTRODUCTION

Several companies have embarked on programs to add voice applications to their work environments. Additionally, much research is being conducted at many of the major universities around the country in the overall area of multimedia information processing. Rather than duplicate the tremendous research efforts embodied in the included references, as well as other non-listed references, the emphasis of similar endeavors in the field of multimedia database design and implementation will be extracted "directly" from the synopsis of the listed reference materials. Where plausible, discussions relating to differences in the approach outlined in this thesis and other systems will be examined.

B. OVERVIEW

Terry and Swinhart in [Ref. 10] thoroughly discussed many of the systems that have been introduced in this area over the past few years. Several comprehensive comparisons between their Etherphone system and other related systems were offered. This thesis shall concentrate on the sound management aspects of such systems.

The Etherphone system uses a structure known as a voice rope. Voice ropes are encryptically recorded voice segments stored as files and maintained in a voice rope database. An entry in the voice rope database contains attributes for the identifier, creator, access-control lists and overall length of the voice rope. By using the identifier-attribute approach (flat table view), only a single database access is required to determine the voice rope's complete structure. [Ref. 10] Our approach is designed to limit the amount of data stored in the database by including certain physical characteristics of the recorded data within a header structure of the actual sound data file.

Lockemann in [Ref. 11] offered a section specifically directed at the implications of voice data management. His work effectively highlighted many of the operations performed by these systems as they relate to the concept of multimedia database systems management. Christodoulakis et al. in [Ref. 12] highlights similar aspects of the multimedia systems that were covered by Terry and Swinhart in [Ref. 10] and by Lockemann in [Ref. 11], but goes on to also compare the suitability of their MINOS system to what they call an "open world" environment in which certain information may not be very well known in advance. MINOS emphasizes the idea of having multiple methods of finding and linking diverse information.

MINOS is an object-oriented multimedia information system that provides integrated facilities for creating and managing complex multimedia objects. MINOS incorporates functions that exploit the capabilities of a modern workstation equipped with image and voice input/output devices to accomplish an active multimedia document presentation and browsing system. [Ref. 12] states that "...queries on the attribute and text part are similar to those allowed by database management and text retrieval systems (conjunctions, disjunctions and parts of words)," and that "...the same query capabilities (on the voice part of MINOS multimedia documents) are allowed for text as for voice." The query specification interface assists the user in interactively specifying a query with the help of menus and some graphics capabilities [Ref. 12]. Voice segments and voice narrations are "dependent" components of a MINOS document and do not exist independently of the document. By contrast, the model presented in this thesis presents sound objects in general as independent entities within the multimedia database.

Systems such as MINOS and the Electronic Document System (built by the BALSAs project group at Brown University) emphasize the ideas of hypertext and information webs. "Webs are links to some other information with relevant context" [Ref. 12]. The

underlying textbook analogy is maintained in that each image or voice object is "dependently" linked to a particular document.

We are interested in developing an overall multimedia information system based upon an underlying relational multimedia database. It should permit the end user the ability to access stored real-world values (i.e., sounds or images, etc.) which represent real-world objects from within the database. The issue of capture and storage of these values remains a driving concern in the ongoing development of this research project. Up to this point there was nothing particularly new or revolutionary regarding our approach to the problem of integrating multimedia objects within a common database. However, there are some rather extraordinary exceptions. Our primary area of ingenuity resides in 1) the introduction and use of an abstract data type of type SOUND; 2) the definition of the operations that are needed to manipulate this multimedia data of type SOUND; 3) the storage of the contents of the sound data object in the database via a description attribute which can be accessed using the extended syntax of a standard structured query language (e.g., SQL); and 4) the storing of physical characteristics of the sound object within the sound object data file inside a file header. This approach has unveiled an area of research well within its earliest stages of development.

This thesis concentrates exclusively on the management of sound within such a framework. Although the final project is yet to be realized, we can see both clear distinctions and comparisons between this and other systems that have already been developed. Both IBM's experimental Speech Filing System (a stand-alone special purpose system), which was operational in 1975 [Ref. 13], and the Etherphone System [Ref. 10] rely heavily, although not exclusively, on the use of telephones to access stored voice data. This differs from our approach since we can manage more than just voice data and since most workstations of the future will have their own sound output facilities. In the event

workstations do not offer self contained sound handling devices, our research is adaptable to the full and complete integration of off-the-shelf products in order to achieve this capability.

Once sound data is captured, access to that data must be available. The Etherphone system can connect individual and arbitrary fragments of voice files to form a "voice rope". Etherphone uses voice ropes to access data, plus a sequence of light and dark bars displayed on the screen to represent intervals of voice and silence. The Experimental Multimedia Mail System of [Ref. 14] provides interactions with the voice data via three types of windows. The "voice editor" is used to create, record, play, select and display voice data. Once access to a particular voice object is obtained, editing can be performed. The "voice display" shows already existing voice data as an energy waveform on the computer display screen. The third window, the "voice buffer", serves as an intermediate storage location for speech segments.

The Diamond Document Store system [Ref. 15] also uses a form of waveform interpretation of voice data for word selection and editing. Waveform interpretation, however, is both imprecise and time consuming. Maxemchuk [Ref. 16] avoids the problem of analyzing the waveform for the beginning and end of words by associating various parameters of the recording (e.g., intervals of silence, playback rate, arbitrary start and stop points, etc.) for the recorded output. We have infused an adaptation of this approach in our research regarding the management of stored sounds within the database. For example, we also provide the user the ability to select specific start and stop points within the referenced sound data object. These points, however, are strictly based on elapsed time from the beginning of the (converted) sound object's data values.

Almost all of the systems reviewed use a file storage system for the management of sound data. This somewhat universal approach, in turn, forms the basis of our data

storage scheme. However, additional unique information is also stored within both the database and the header of the data file to facilitate ease of query processing. The storage structure for the information is never revealed to the user. The Etherphone system reduces duplication of voice fragments by maintaining a list of pointers to the individual fragments in the voice tracks.

The above systems have established the foundation for the exploration of the management of voluminous, shared data among distributed and heterogeneous workstations [Ref. 10]. The techniques presented in this thesis are designed to build upon previous work and should be applicable and beneficial in the growing area of multimedia database systems design.

V. THE SOUND DATABASE AND INFORMATION SYSTEM

A. SOUND MEDIA MANAGEMENT

The integration of multiple media into the personal workstation and other computers offers tremendously increased potential for improved productivity across a wide range of applications and environments. Fortunately, sound and other non-textual information media can now be effectively and efficiently stored in an integrated information system. This is due primarily to the ever increasing improvements in hardware and the similarly decreasing costs of memory and storage.

A multimedia information object may contain several attribute media (i.e., text, graphics, sound, images). Each, in turn, may be composed of a number of highly specific and intrinsically exclusive attributes. Conceptually, each medium object is an integral part of the entity object (i.e., record field, tuple attribute). Each object is logically represented as a complex tuple, although the actual storage of the various data values may exist in several forms (i.e., hierarchical, normalized, unnormalized). By properly managing stored sound in the database, a further increase in the accessibility of real world information can be made available to the user.

There are several distinct advantages that a typical database integrated with sound can offer:

- The unrestricted use of voice narratives and annotations in interactive applications.
- The recording and storage of naturally occurring, though textually undefinable, sounds.
- The retrieval of previously stored sound data.
- The sharing of sound data files among various users.

Many of these features are not exclusive to the sound management arena, but can be found in traditional text oriented databases and file servers as well. The characteristics of sound, however, are significantly different from those of text. Sounds cannot be easily captured, stored or manipulated. A number of the more basic characteristics with regard to the handling of sound (e.g., frequency, sampling rate, etc.) were discussed in Chapter II.

As an example, consider the characteristic of size with respect to sound. A page of text would take about two minutes to read if read out loud. The capacity needed for storage of the written text is rather nominal (@ 3.5 Kbytes). Yet the storage required for the same spoken two minutes of text, once digitized and recorded, would require several orders of magnitude more (@ 1Mbyte) than its equivalent written counterpart. This brings into focus another important aspect of sound management which needs to be considered. Specifically, the handling of sound may also require special devices to enable sound recording and playing. And finally, the real-time nature of sound imposes stringent requirements upon its capture, presentation and synchronization. These differences dictate that special methods must be employed when managing sound in a multimedia information system.

B. A MULTIMEDIA ARCHITECTURE

An important issue of multimedia database management is the design of a suitable data model. The structure of multimedia data objects is generally more complex than that of the more familiar text-based data objects. Though it may be possible to provide to the user a seamless interface for interactions with the database, the functionality of the underlying DBMS must become much more highly specialized.

Although a great deal of research has been devoted to data modeling, there remain several problems relating standard database modeling techniques to a more generic model for multimedia database objects. The multimedia object may be composed of a collection

of components--that is, specific medium objects--that differ in the type of multimedia information contained within.

The user interface, another underlying concern, must always be considered in designing the multimedia database. For example, sound management in a multimedia database must permit the user many of the obvious functionalities that readily come to mind when dealing with sounds of any type. This functionality includes identification, capture, storage, retrieval, analysis and manipulation. By grouping related functionalities (or, in a sense, characteristics), we can describe the *logical structure* and the *physical structure* of the multimedia object [Ref.7]. The *logical structure* determines how logical components, such as basic data elements, are related to the contents of the object. The *physical structure* determines how the physical components, such as storage characteristics and presentation devices, are related to the contents of the object.

The structure of a multimedia object, however, is best described in terms of its conceptual components. We shall view the *conceptual structure* as the "big picture" or expanded view of the logical model. This convention is adopted exclusively for our use and therefore should not be viewed as a universal declaration of the terms. The *conceptual structure* is important since it is far more meaningful to the user. A formal data model can be used to define an information object's conceptual structure. Such models provide powerful abstractions for describing the semantics of data in complex applications. These abstractions shall be used in describing the conceptual structures of the SOUND information object. The SOUND information object (or just SOUND object) is the primary multimedia object upon which we shall focus our discussions throughout the remainder of this thesis.

Figure 4 depicts the conceptual makeup of a multimedia object. Each multimedia object may have one or more multimedia components. Each variation of a component may be

viewed as creating an entirely new multimedia object or just an extension of the original. It is entirely up to the user as to which conceptual view is more meaningful.

A data model should allow a very natural and flexible definition and evolution of the schema that represents the composition of and the complex relationships among distinct parts of a multimedia object. It should also provide a means for the sharing and manipulation (i.e., storage, retrieval and transmission) of multimedia information.

Queries may have conditions on both the conceptual (i.e., a SOUND object) and the content structure (i.e., the "real-world" representation) of a multimedia object. We refer to the content structure (not depicted) as the textual description of the object which is stored in the database and is always available for use. For example, when conducting an SQL query on an IMAGE attribute of a multimedia object, conditions on the object's conceptual structure are satisfied by locating the conceptual components (obtainable from internal functions) specified in the condition. Because SQL does not support the IMAGE object directly, *a translation of the query in terms of the content structure may have to be performed.* At the end user's level, the level above the programmer's, this would possibly involve the use of interactive menus and some graphics (i.e., icons) manipulations designed to call certain functions. These functions would perform the low-level operations required to satisfy the user's query. If graphics are used, they should serve as abstract representations of the desired end result.

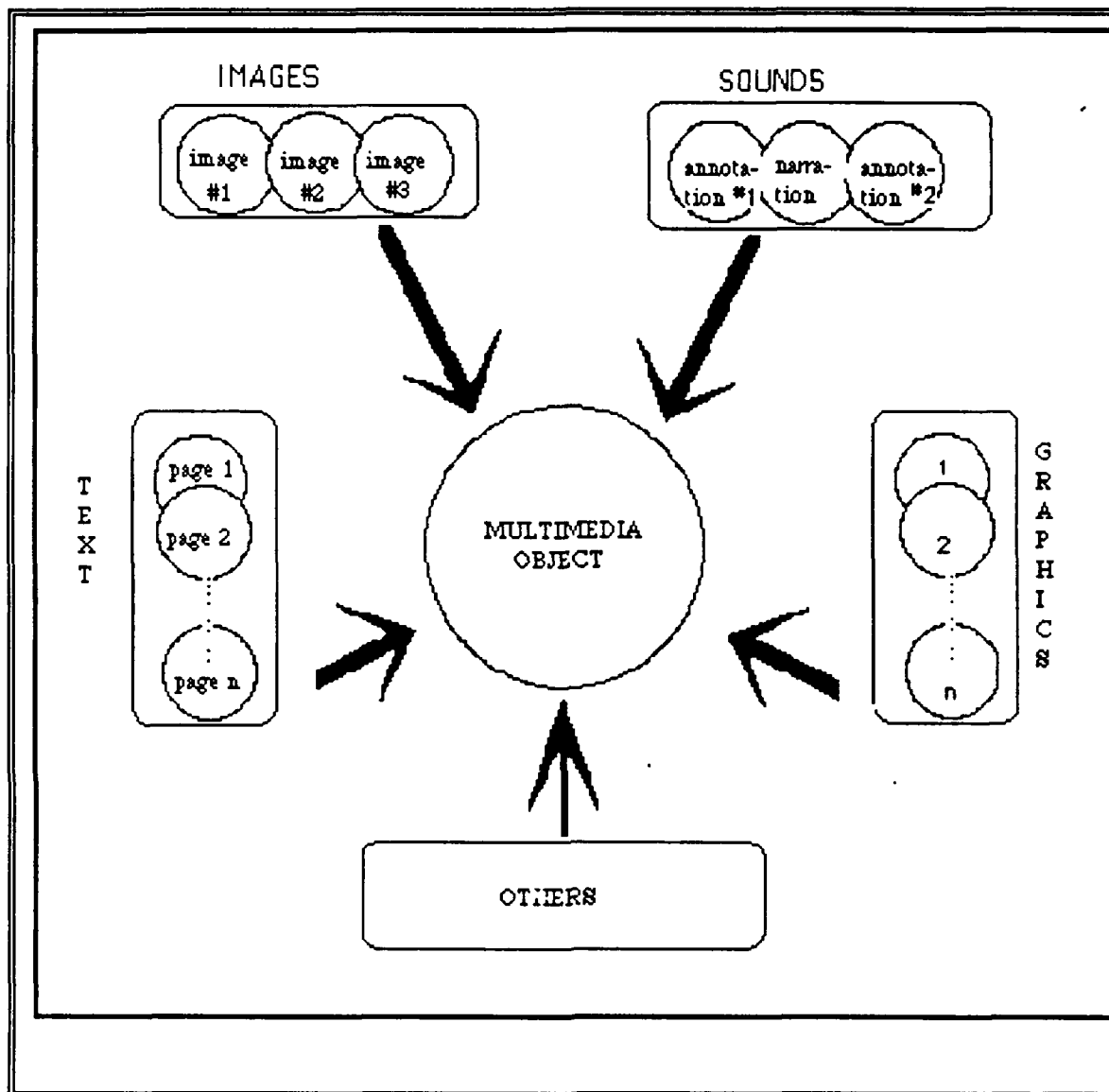


Figure 4. The Conceptual Model of a Multimedia Object

Figure 5 shows the relation of the conceptual components of a multimedia database system. The user is shielded from the specific actions of the multimedia database system

and should remain unaware of the activities involved in the presentation of multimedia information. The actual information retrieval process remains intrinsically invisible.

At a much lower level, the *query processor* level, the query would be converted into suitable database function calls to the *supervisor* (or application level). Any preprocessing requirements of the query would also occur at this level. The *supervisor* would provide a sequence of function calls to the appropriate data handler (media manager). The *data handler* would provide the necessary interface to hardware where the actual data is stored, generally in files, for retrieval and presentation. The data model of the multimedia information system, when presented in this fashion, can be seen as primarily a media management system with a few highly important added features to enhance its functionality. In particular, the multimedia data model offers functions for administrating the data, semantically linking the data, synchronization and coordination of the data and data storage. By making the system open-ended, new applications and data types may be added when desired.

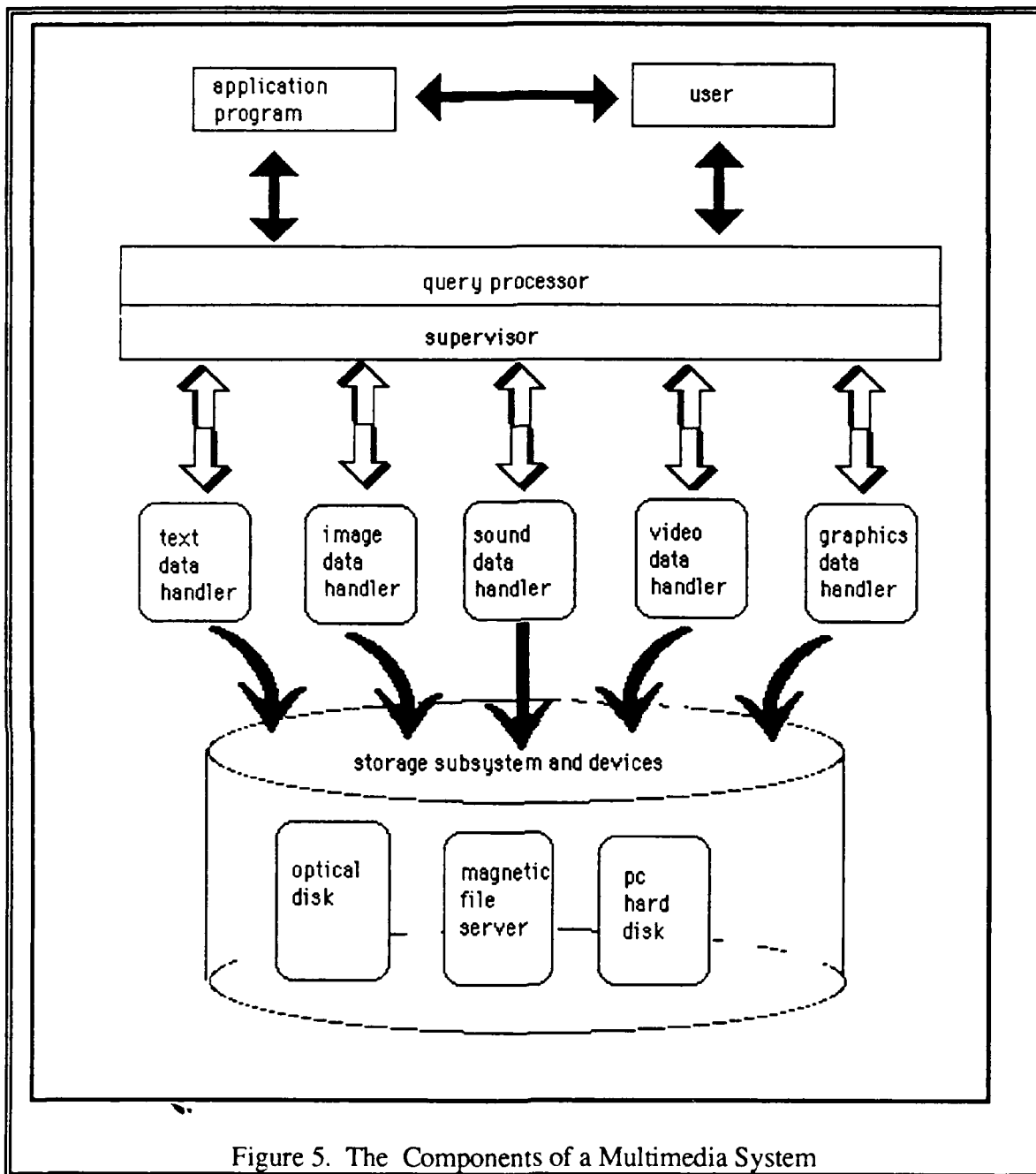


Figure 5. The Components of a Multimedia System

C. THE SOUND DATA TYPE

1. Sound Data Organization

The foundations of the approach to unformatted (or unstructured) data as applied to images in [Ref. 1] will serve as the basis for the establishment of the SOUND data type. For clarification, the reference to unformatted data simply means that the meaning of its values are not associated with the actual processing of the database. The traits and characteristics of the data need not be known to the DBMS when the various data forms are stored and retrieved. Many of the ideas expressed for the IMAGE data type can be directly applied to sound. The crux of that approach will be skeletally duplicated here for clarity and completeness, but with a directed emphasis towards the effective management of sound.

The relational database model shall be adopted when discussing multimedia database query techniques. This allows for the conceptually familiar flat table view of the attributes relating to a multimedia object. Each tuple of the multimedia object will consist of several medium related attributes, such as graphics, images and sounds. With this idea in mind, sound, as with images [Ref.17], can be regarded as an *abstract data type* with its own set of operators or functions. The specification of our structured abstract data type includes 1) a component element type (SOUND); 2) a structure that relates the component element values (database relational table); 3) a domain of allowable structure values (encoded sound data); and 4) a set of operations on the values in the domain (sound handling functions).

Because of the rather primitive capabilities available regarding the semantic analysis of a waveform, this rudimentary approach shall be consciously avoided. Although certain characteristic features may be nice to have, our concern centers solely around the actual semantic contents of the sound source. Very little information can be obtained from the

merely graphical depiction of a complex waveform without some prior knowledge and understanding of the contents of the sound the waveform portrays. Waveform analysis is best performed by experts in the field of acoustics and should not be a necessary skill required of the typical end user. To circumvent this exceedingly low level, complex and inherently tedious operation, we have abstracted the contents of the sound data (and other non-textual forms) into descriptive words or text. By storing the "equivalent" textual contents of the sound as an attribute of the sound data relation, we can achieve a tremendous improvement in database access and query performance. The content is strictly determined by the description assigned by the human user during the storage of the sound data object, such as the name of the speaker, the nature of the speech or the subjective description of the sound itself.

Meyer-Wegener et al. [Ref. 1] describe this abstraction process as consisting of three parts: raw data, registration data and description data. *Raw data* is the actual bit sequence storage of the data. With sound, for example, this sequence could be the byte sized ADPCM digitally encoded samples. *Registration data* is the data related to the physical aspects of the raw data. This data incorporates the encoding algorithm needed by the presentation device to display the raw data in a high level "user understood" form. Examples would include the color intensity and the colormap for an image, and the sampling rate and encoding technique for a sound. *Description data* relates to the contents of the multimedia data captured (entered) by the user. It is a natural language (i.e., English, Dutch, Spanish, etc.) description of the contents of the data. For a short voice segment, for example, this could be the actual words that were spoken, as well as the time, the place and the name of the speaker. This data will be used for contents search during the multimedia database query process. As with any type of abstraction process, a loss of

information will be present relative to the original source. Such losses are an unavoidable, though naturally occurring, phenomenon.

2. The User's View

Information systems should be designed with the end user in mind. Multimedia information systems are no exception. Users must be able to gain access to data in a manner which is most useful to them. The actual operation of the underlying database should be entirely of no consequence to the user, although the "hidden" database management processes should be relatively fast and accurate if they are to be considered effective. Keeping the previous two sections in mind, the time has come to discuss the user's view in the management of sound data within the framework of the multimedia database system.

Managing sound in a multimedia information system requires close ties with the presentation aspects (i.e., output devices or playback speeds) of the actual sound data. Clearly, such data management concerns encompass more than the mere storage and retrieval of information as performed by the multimedia database. The abstract data type SOUND will be used to model the actual occurrences of the sounds available in a multimedia database system. This type allows the specification of certain intangible sound related properties which can be managed by a multimedia DBMS.

In some systems, the information presentation process may directly influence the structure of the underlying database and the database management system involved. On the other hand, a well-defined DBMS may require that the multimedia data be stored and retrieved in a "preset" or fixed manner. That is, the data may have to be "transformed" before it can be fed to an output device for presentation. Other criteria may dictate restricted random access to specified sub-parts of the data. Such considerations are crucial

to the development of multimedia information systems, but the details of their implementation should always remain "invisible" to the end user.

Several speculative methods pertaining to the user interface have been presented in related works. Many of these works are discussed in Chapter IV. The unrealized end user's interface of this thesis related research remains an important part of ongoing studies. Different applications of the DBMS may dictate different user interfaces. In this thesis the actual structure of the end user's presentation and interface will not be discussed. The programmer's interface, on the other hand, will be examined since its functionality relates directly to our development of the prototype.

The typical end user is not expected to know either the format or the structure required of an appropriate multimedia database query. The details of the "how" remains within the realm of the underlying application program and multimedia DBMS. The end user should need only to specify "what" is desired, possibly via a set of menu commands and/or icons.

By comparison, the programmer's interface requires the use of a structured query language, like SQL, to interface with the multimedia DBMS. The programmer is singularly responsible for the formulation of the appropriate query. This level also incorporates a lower level of "what" must be done. It seems more appropriate and considerably more convenient to make the components of a SOUND object accessible through a series of functions, rather than through a complex, variable length record. By introducing a series of sequential function calls, the desired data can be retrieved from the database and presented on the appropriately specified device.

Different functions can be defined to produce different output types for the components of the SOUND data type of the multimedia information object. Additionally,

low level editing features, such as the interleaving of portions of different SOUND objects, can also be accomplished through the use of function calls.

Consider, for instance, the input function

**CONSTRUCT_SOUND(filename, size, duration, encoding,
samplerate, resolution, description)**

which creates a database entry for a sound object denoted by the more general sound object identifier, **filename**, and constructed with the following input parameters: **size**, **duration**, **encoding**, **samplerate**, **resolution** and **description**. For purposes of this thesis, the **filename** represents a unique file name. Note that both **size** and **duration** can be used as optional parameter specifications which could denote either a fixed size in bytes or a fixed duration in some unit(s) of time (i.e., seconds). The **description** parameter would be used to describe the contents of the sound and any other relevant information pertaining to the sound being constructed.

A unique SOUND object will be denoted by a two attribute relation scheme consisting of

SOUND_OBJECT = (S_ID, S_SOUND).

It may be helpful to view SOUND_OBJECT as a relational table of the form

SOUND-OBJECT

S-ID	S-SOUND
------	---------

Here S_ID is a unique identifier of the SOUND_OBJECT and S_SOUND is the relation (at least conceptually) which stores all of the attributes or characteristics of a unique SOUND_OBJECT which are accessible through a series of function calls. The SOUND_OBJECT relation is created using SQL syntax of the form:

CREATE TABLE SOUND_OBJECT

```
(S_ID      integer2,
S_SOUND   SOUND)
```

where **integer2** denotes a long integer and **S_SOUND** is of type **SOUND**.

A transient SOUND value is constructed that cannot be assigned to program variables (e.g., a rendition of the Gettysburg Address). However, its characteristics as denoted by the parameters of the CONSTRUCT_SOUND function above (i.e., filename, size, duration, encoding, etc.), can be stored in the database and used in INSERT and UPDATE statements of the query language. This approach permits the retrieval and presentation of a single value of type SOUND. An example using the query language SQL could be generated as follows:

UPDATE SOUND-OBJECT

```
SET  S_SOUND=CONSTRUCT_SOUND(filename,&size,  
                                &duration...)
```

WHERE <optional condition>;

Another example could be:

INSERT INTO SOUND-OBJECT

(S ID, S SOUND)

VALUES (3212,CONSTRUCT SOUND(filename,...));.

But what if certain attributes or characteristics of type SOUND were needed? By extending the aforementioned concept even further, other sets of functions could be incorporated to do just that. Each would return a specific value. These values could then be assigned to certain program variables for later use. This set of functions is available to the user and are described as external functions, such as:

SIZE (SOUND attribute): integer;
DURATION (SOUND attribute): float;
ENCODING (SOUND attribute): integer;
SAMPLERATE (SOUND attribute): integer;
RESOLUTION (SOUND attribute): integer;
DESCRIPTION (SOUND attribute): char string;

etc.

Different functions could be defined to produce different outputs, such as a general edit function which would permit the selection of segments of a **SOUND** type or allow several segments to be combined. A representative SQL statement which exemplifies the return of a database value can be seen in the following example:

```
SELECT      SIZE (S_SOUND)  
INTO        :var1  
FROM        SOUND_OBJECT  
WHERE       S_ID = 217;
```

Here, the size of the sound data object with **S_ID = 217** is retrieved from the **SOUND_OBJECT** relation via the **SIZE** function and assigned to the program variable **:var1**.

The use of description data within the database could greatly aid in the retrieval of certain **SOUND** objects. Its applications with regard to the type **SOUND** are identical to those described by Meyer-Wegener in [Ref. 1] for the type **IMAGE**.

PLAY_SOUND, a function which permits the retrieval and presentation of a **SOUND** object from the database can now be more formally introduced as:

```
PLAY_SOUND (S_SOUND),
```

where S_SOUND contains all of the stored characteristics of that SOUND object. The appropriate output device is either implicitly designated by default or else can be explicitly designated as a condition of the query. We shall always assume a default output device for purposes of this thesis. By enabling the retrieval and storage of certain attributes or characteristics of the SOUND_OBJECT into program variables, SQL queries can be even more functionally employed. An example of such a query would be:

```
SELECT      PLAY_SOUND(S_SOUND),DESCRIPTION(S_SOUND)
INTO        :var1, :var2
FROM        SOUND_OBJECT
WHERE       IS_INCLUDED_IN(S_SOUND, "grey whale")
```

This query would retrieve the unique identifier of the SOUND it located from the SOUND_OBJECT database which had a description containing the term "grey whale." Once located, the sound data would be routed to the output device and the stored sound would be heard. For clarity, we've taken the liberty of a rather syntactically casual condition clause to simply point out how these functions can be used. Here, the program variable :var2 would represent the description attribute of the SOUND object which contains the term "grey whale" and the variable :var1 expresses the boolean condition signifying the success or failure of the play process.

If a specific portion of the SOUND object were desired, another function, PLAY_SEGMENT, for example, could have been used to designate a specific start and stop location within the SOUND object file. If more than one SOUND object were desired, a LINK_SOUNDS function could have been used instead of PLAY_SOUND. Similar functions based on the PLAY_SEGMENT and LINK_SOUNDS functions could be included which would create a new SOUND object file based on the input parameters. For example, a CUT_SEGMENT function could create a new database entry by removing

a specific portion of the data from the input object file. A `CONCATENATE_SOUNDS` function could add another entry into the database by concatenating two input `SOUND` objects to produce a new object file as a result.

It should be becoming clearer that we can achieve tremendous flexibility in the management of sound objects through the use of such functions. To summarize the above discussions, an extensible list of basic functions pertaining to the manipulation of sound data has been collected below. These functions are designed to operate on attributes of the `SOUND-OBJECT` domain. In addition to the general operators, certain functions that are specific to our implementation are also included. These are denoted by the inclusion of the term, "ANTEX." In general, most generic internal function may have an "ANTEX" counterpart. Since our specific implementation is written in the "C" programming language, the constructs of this language will be used to explain the input parameters and output results.

Most of the function names are descriptive of the operations which they perform, perhaps with the exceptions of the functions `DATA_ONLY_FILE` and `IS_INCLUDED_IN`. The `DATA_ONLY_FILE` creates a copy of a data file in PCM format minus any header information which may be stored in the file. Its `ANTEX_FILE` counterpart would produce an ADPCM encoded data file and would also not include any header information. The `IS_INCLUDED_IN` function searches the database to determine whether or not a stored `SOUND` object contains the descriptive string pointed to by `*char.` For now, this is merely a pattern matching function which operates on the description attribute and returns a boolean result. The "side effects" are simply the stored sounds being played via an output device.

FUNCTION NAME	INPUT (arguments)	OUTPUT (result)
CONSTRUCT_SOUND	(see above)	SOUND
ANTEX_SOUND	"	SOUND
SIZE_OF_OBJECT	SOUND	long integer
SAMPLE_RATE	SOUND	integer
TYPE_ENCODING	SOUND	integer
DURATION	SOUND	float
BITS_PER_SAMPLE	SOUND	integer
DATA_ONLY_FILE	SOUND, *char	integer (+side effects)
ANTEX_FILE	SOUND, *char	integer (+side effects)
ADD_DESCRIPTION	SOUND, *char	SOUND
REPLACE_DESCRIPTION	SOUND, *char	SOUND
DESCRIPTION_LENGTH	SOUND	integer
DESCRIPTION	SOUND	*char
PLAY_SOUND	SOUND	boolean (+side effects)
ANTEX_PLAY	SOUND	boolean (+side effects)
PLAY_SEGMENT	SOUND, float, float	boolean (+side effects)
CUT_SEGMENT	SOUND, float, float	SOUND
LINK_SOUNDS	SOUND, SOUND	boolean (+side effects)
CONCATENATE_SOUNDS	SOUND, SOUND	SOUND
RECORD_SOUND	--	SOUND
ANTEX_RECORD	--	SOUND
IS_INCLUDED_IN	SOUND, *char	boolean

Again, the "ANTEX" functions are model specific reflections of the more generic sound data manipulation functions. With this operational backdrop in mind, an implementation of the SOUND abstract data type can now be more formally described.

3. Implementation of the Abstract Data Type

For a concept of this nature to be of quantifiable value, it should be (preferably) implementable at some fixed or basic level. The following implementation is based on the concept of relating attributes and objects as a means for retrieving previously stored SOUND objects from within a multimedia database system.

As previously noted, a multimedia information object will consist of one or more computer manipulated media types. In the relational view, each representative media would be stored as an attribute of the parent multimedia object. Each attribute would have characteristics that are rather unique, including the requirements of special presentation and (possibly) storage devices. For the abstract data type SOUND, we proposed that the basic

composition of the attribute S_SOUND of the SOUND-OBJECT relation contain the following fields or characteristics:

- s_filename	char[64]	/* a unique name */
- s_size	long integer (bytes)	/* of data file */
- s_duration	float (seconds)	/* can be computed */
- s_encoding	[PCM, ADPCM, LPC, etc.]	/* DSP algorithm */
- s_samplerate	integer (8000, 16000)	/* # samples/sec */
- s_resolution	integer	/* # bits to digitize */
- s_description	char[500]	/* text description */

where the "s_" prefix represents a sound related characteristic of the SOUND object. In our approach, each SOUND object represents a different sound data file.

Unfortunately, a considerable amount of the characteristics information could be lost if the actual data is ever separated from the database. One method for reducing the amount of data being stored in the database is to store specific characteristics within the data file itself. This is the approach which we have used in establishing our model. Specifically, a header containing the sound data object's size, samplerate, encoding, duration and resolution has been stored in the data file along with the data. This reduces the items stored in the database to only the filename and the description. Although this information could also have been stored within the header, it is considered far more useful as a linked pair of a distinct database entry. We are assuming that a search based on the description attribute will be a fairly frequent event, as opposed to similar searches based on the size, duration, etc. of the SOUND object. This process is consistent with that which has been successfully implemented in the related work on image objects.

The information stored in the database will permit ease of access and better query processing of SOUND objects. Note, however, that not all of the characteristics of sound

are explicitly stored in the database. These include such perceptual items as pitch and intensity, as well as the physical characteristics of the individual amplitudes and frequencies. Since our aim is to develop a model for the capture, storage and management of sound data, the additional information mentioned is somewhat unnecessary. Moreover, complex algorithms such as the Fast Fourier Transforms (FFT) series would be needed to extract, for example, the multitude of frequencies residing in a typical complex waveform. Such a process is clearly undesirable. Furthermore, it does not support our need for multimedia information access.

The *s_filename* will permit a SOUND object to be uniquely identified in the database. The actual sound data (in the form of files) is typically linked to a particular object tuple of the complex multimedia relation. Thus the SOUND attribute of the multimedia object may be individually queried.

To avoid unnecessary duplication of the data within main memory, greater flexibility is achieved by feeding the data file directly to the device driver and the output device. The *s_filename* is passed as a parameter via a function call to the sound data handler (typically a device driver). An example of this can be seen in the previously introduced ANTEX_PLAY function where the name of the file to be played is passed as a parameter.

Avoiding intermediate storage and unnecessary copying of massive data is a very important design issue of multimedia databases and one which received critical awareness in the design of our sound handling prototype. The prototype uses a background interrupt and an on-chip buffer for piping the data directly to the output device without main memory processing or storage.

The *s_size* attribute denotes the size of the SOUND object (in this case, the actual sound data file) in bytes. *S_size* is directly related to *s_duration*. The *s_duration* of the

sound is the amount of time in seconds and hundredths of a second it takes the file to play from start to finish. This figure is inclusive of all sounds recorded during the original capture sequence, such as white noise or silence. *S_duration* is based on the sampling rate and the rate of playback. Compaction algorithms may differ between machines, although most computer related applications currently use ADPCM because of its reasonably high quality reproduction with respect to both voice and music. The *s_samplerate* is needed to alert the sound data handler (device driver) of the correct speed in which the stored sound should be played.

The *s_encoding* attribute alerts the device driver of the type of algorithm used to convert the original analog signal into digitally encoded data. Should the decoding algorithm differ from that which was used to encode, there will be a general loss of quality in the audible output. Since almost all encoding is performed by DSP chips, it is vital that the algorithm employed in the decoding process be compatible.

The number of bits used per sample is known as *s_resolution*. Most speech digitizing chips have found excellent resolution in the use of eight bits per sample. However, depending on the architecture of the machine, even greater signal fidelity can be captured by increasing the resolution of the sample. We define *s_resolution* in terms of the digitizer (codec) rather than in terms of the compression algorithm used to encode. Unfortunately, extraction of this physical characteristic is exceedingly difficult with the details remaining to be worked out.

Since waveform analysis should not be a required skill of the user, the *s_description* attribute will be used to determine the semantic contents of the SOUND object. This will permit browsing the contents without actually having to play the stored sounds in order to determine their contents. By storing the textual representation of each

SOUND object in the database, a considerable savings in database storage requirements and query access time is achieved.

VI. DESCRIPTION OF A SOUND MANAGEMENT PROTOTYPE

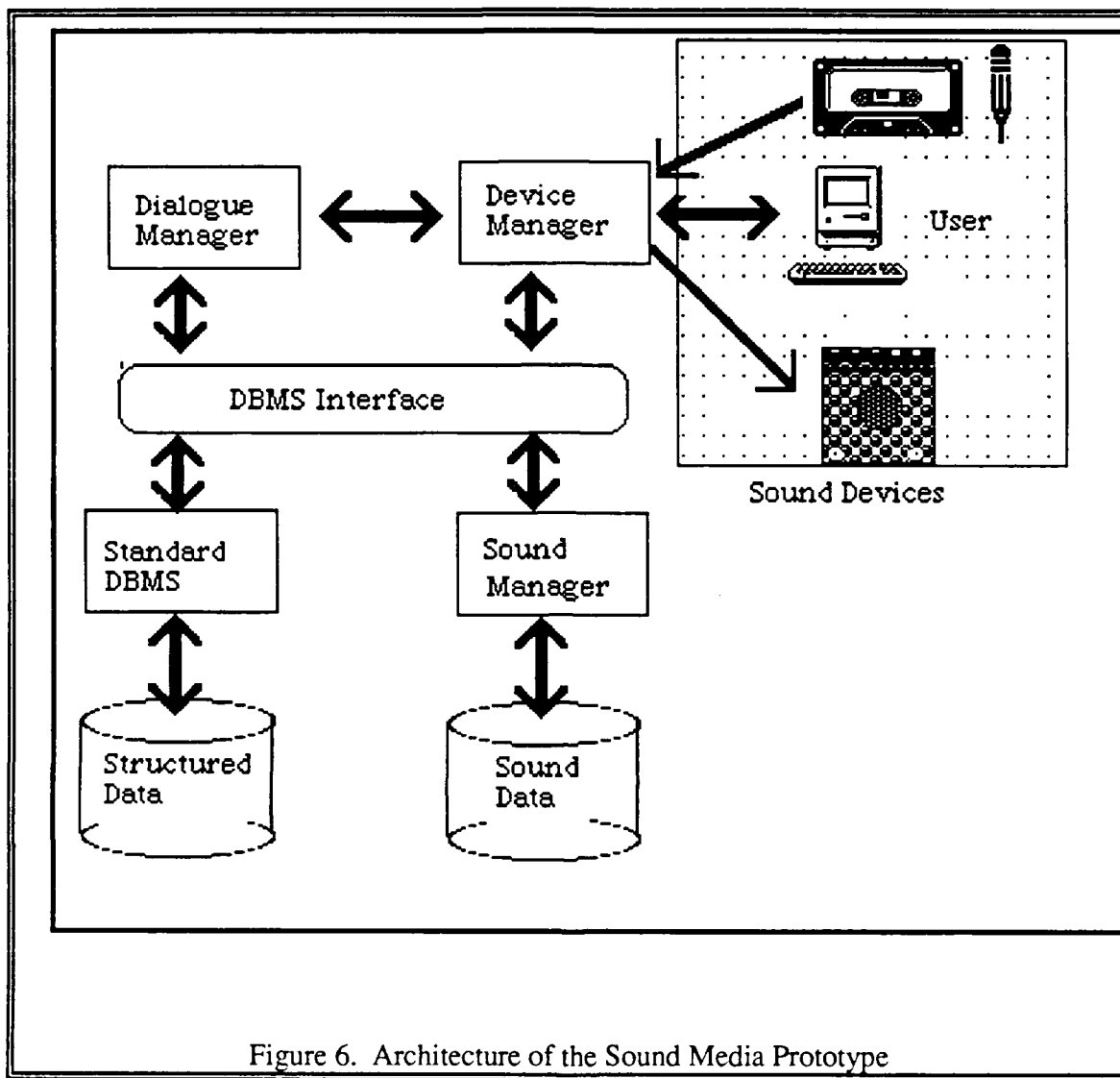
A. ARCHITECTURE OF A PROTOTYPE

The prototype is intended to provide extensibility to the current relational DBMS by adding the objects of type SOUND to the database. In an effort to maintain our "off-the-shelf" technology approach, the functionality of the sound management system is built around the commercial relational DBMS "INGRES." The host language is standard "C" using the embedded SQL statements and commands provided by "INGRES."

Figure 6 reveals the high level view of the architecture. It is noticeably similar to that which was employed for the discussion of the IMAGE DBMS [Ref. 1]. More distinction, however, has been added to the concept of the sound devices.

The *dialogue manager* serves as the interface between the device manager and the database management system. It can be viewed as the heart of the system, performing all exchanges of data and the employment of the various input and output devices. The *device manager* serves as the actual physical link between the main program and the various devices. It is composed of the various software drivers needed to activate or interact with the hardware. Figure 6 shows the different types of sound devices that the device manager may need to manage: namely, input devices such as microphones, audio players or vcr's, etc., computer based sound generation chips, and output devices, such as amplifiers.

The *DBMS interface* implements the query language integration within the system with respect to the various sound attributes. The DBMS interface is a general purpose set of functions whose level of integration is not exclusive to sound. The *sound manager* is a set of functions which perform the actual storage of the sound object, as well as performing other administrative activities such as generating a unique identifier or determining the size of a sound data file. These functions were described in general terms in Chapter V.



Although not depicted, for the sake of professional clarity we can also assume an arrow also exists between the *device manager* and the *sound manager* as there exist sound related functions in which the DBMS interface is never invoked.

The next section will discuss the specific hardware requirements necessary for the development of the prototype. The coding of the various transformation functions needed to properly query a database integrated with sound remains an area for follow-on research.

B. MODEL SPECIFIC EQUIPMENT

The requirement for high resolution graphics and image capabilities led to the SUN System workstation as a target machine for the full research project; that of building a multimedia database system using relational database technology. Unfortunately, the lack of input/output audio facilities for the SUN 3 workstation required pursuing other areas of implementation for the SOUND object data handling. Incidentally, Digital Sound Corporation's DSC-200 Audio Data Conversion System is a unique peripheral subsystem specifically designed for optimum performance on the SUN 2/3 Workstations. The total system cost, however, was a prohibitive factor at this stage of our research. A reasonable alternative with similar or better operational functionality was needed.

We were able to find the necessary functionality through Antex Electronic's Model VP620E PC Compatible Digital Audio Processor. This highly versatile board performs real-time A/D and D/A conversions, with corresponding encoding and decoding of the digitized signals. Using this, in conjunction with other peripheral devices and software components, provided a platform for evaluating the management of sound in a multimedia database and information system environment.

The equipment employed in our model specific prototype is listed below.

Hardware:

- IBM PC/AT/386 compatible with 20MB internal hard disk
- Antex VP620E plug in Audio Processor board (installed)
- Standard cassette deck with min 1VRMS audio output port
- Plug in microphone (standard 1/4" jack)

- Audio Amplifier (standard 1/4" input connection)
- Various connection cables

Software:

- Microsoft 'C' with standard libraries
- INGRES DBMS language (with embedded SQL)
- Audio driver routines for Antex VP620E Audio Processor board
- Driver interface functions
- Query language transformation functions (future research area)

Input signals are taken directly from the microphone or the output port of the audio cassette player into the standard 1/4" input port of the VP620E sound board. Microphone signals must be at least 1VRMS in order to be received above normal threshold noise. The sound board filters and samples the incoming analog signal at either 8KHz or 16KHz (software selectable). Each sample is converted into an 8-bit digital number by the codec. This number is then encoded (compressed) using ADPCM by the DSP chip which results in a 4-bit sound data sample. Once in this form, the sound data files can be manipulated via the sound driver software and the associated function calls to it.

Output is achieved by reversing the capture process. The 4-bit sample is decoded by the DSP chip into an 8 bit sample, then reconverted by the codec into an energy level representation which is routed via the output port of the sound board to the amplifier. The amplifier then translates the energy levels received into frequency responses which causes the speaker(s) to vibrate, thus reproducing the stored sound.

The Antex VP620E comes with a demo program which permits testing and evaluation of the various sound management capabilities needed for the future high level development of the prototype. This model also incorporates one of our long range desires. That is, the sound presentation can be performed in a background mode, allowing the user to perform

other activities on the computer while a sound is being played. This is accomplished by piping the sound data directly from the database to the DSP chip of the sound board without first copying the data into memory. We are able to string several sound files together in this manner without a corresponding loss in user productivity.

Sound stringing can be performed by the user in one of two ways. First, the user can specify which SOUND objects are to be played in the sequence desired. These will be played as requested with little to no time lapse between selections. The other method is to specifically request segments of a SOUND object or SOUND objects to be played. For this option, the user must specify both the SOUND object, the start time in seconds and hundredths of a second, and the end time for each object. The default for each of these is the beginning and the end of the SOUND object, respectively.

C. IMPLEMENTATION CONSIDERATIONS OF THE MODEL

At this stage, the prototype has demonstrated strong support of our thesis research. We have been able to achieve the capture, storage and retrieval of sound data. The foundation for the required internal database manipulations has been presented and discussed and is currently in the process of being formally implemented. The SOUND data model developed in Chapter V provides a highly interactive manner for manipulating stored SOUND objects in a database.

However, the prototype implementation has also exposed a few of the false impressions that were being carried with respect to the manipulation of SOUND objects. The most prevalent of these encompasses the erroneous concept of hardware independence of the encoding algorithms. The Antex VP620E uses an ADPCM encoding of the digital signal emerging from the codec. If other machines are to be used, namely the SUN Workstation, the system must be able to successfully decode the encoding algorithm of a

sound file which has been imported from another source. Since almost all current digitizers perform follow-on encoding in hardware, this could pose a serious problem.

In addition to the portability problem of the encoded sound files, additional peripheral equipment may also be needed in order to support presentation features of the multimedia information system. Without a means for input (capture) and output (play) of the SOUND object, the functionality of the underlying DBMS will be of no avail, regardless of how ingenious it may be implemented.

Issues of compatibility and availability of highly specialized equipment (i.e., plug-in sound boards or sound generating chips) could severely hamper the power of the multimedia database. The ability to determine the sampling rate used on a sound data file and then to emulate the reproduction of that rate on a sound presentation device must also be taken into consideration when handling sound data. When this type of information is included in the header of the data file, it is much easier for the sound object to be properly reproduced. By adding a header to the data only file created by the Antex VP-620E, we've circumvented a severe limitation of our model. Without this header, file specific information must be stored in the database at the time of file creation; otherwise it may be lost. This information must then somehow be transferred along with the data file if the data is to be of any meaningful use to the receiving user.

The ability to string various SOUND object segments together yields a high degree of flexibility for the user. It should be noted, however, that the sampling rate must be the same for SOUND objects that are to be strung together. Sampling rate incompatibility, as could be expected, results in an unintelligible output.

VII. SUMMARY AND CONCLUSIONS

A. REVIEW OF THESIS

The 1980's has ushered in a new era of information management in the form of the computer. Our thesis set out to unravel the mysteries surrounding the management of sound data within a multimedia database management system. Several key questions were introduced in Chapter I. These have now been fully answered through the research milestones outlined in the chapters which followed. A synopsis of the results are included below.

For our work, sound data is exclusively stored in files. Each file is referred to as a SOUND object where collections of these objects form SOUND relations. By extracting out the semantic contents of a SOUND object into words, then storing this description within the database, we were able to introduce queries which could be used to locate specific types of SOUND objects. Additional information can be obtained by storing each SOUND object through the use of a unique identifier. Along with the recorded sound data, a header of related attributes and characteristics pertaining to the sound data is also stored inside each file. These attributes denote the size, sample rate, encoding algorithm, duration and resolution (or bits-per-sample) of the sound data. This information is accessible through a set of function calls specifically designed to extract requested information from a SOUND object.

B. APPLICATIONS

The ability to manage sound in a multimedia database offers tremendous advantages to the user. The user's concerns generally center around the presentation aspects of the

multimedia information system. The user typically knows very little about the integrated components of the DBMS. Therefore from a users view, several applications can be suggested for the use of this powerful information medium.

One of the premier areas of multimedia information uses is in training. This is of particular importance to DoD agencies. By stimulating multiple senses at a time, the ability to process greater volumes of incoming information is increased, which in turn reduces training time. And since time is money, multimedia information systems offer a positive money saving alternative to single medium standard training programs.

Security is another area which benefits from the ability to manage multiple media within a database. The use of voiceprints have already found a niche in the world of access control. A multimedia DBMS incorporating SOUND objects could be used to selectively attach sound bites to different passwords or documents.

Information retrieval is enhanced through the use of multimedia information systems. When sound is used, multiple personnel can take advantage of the audio information resulting from the management of sound at a single workstation. In a closed work environment with limited output devices, this could effectively reduce the need (depending on the application) for duplicating or sharing large data files between users.

The real-time capture of sound signals at remote or isolated sites (i.e., an underwater acoustic signal from a submarine) can aid in the proper identification of unidentified signals. This is accomplished by comparing the captured signal with signals that have been previously stored within the database. Taking advantage of this capability could offer tremendous savings in time and effort during remote operations where support is either limited or lacking.

Perhaps the most widespread use of multimedia information has occurred within the area of office automation. Extending inner office memos, mail and message systems with annotations and narrations has meant greater overall productivity for all employees involved.

C. FUTURE RESEARCH AREAS

Much work remains in the area of query transformations. A foundation has been established for the design and implementation of functions which will transform a user's query entered as an embedded SQL request into a standard "C" program which replaces each embedded statement with a corresponding set of function calls to the appropriate SOUND object relation(s) of the database. Appendix A provides a brief insight into how this process should occur. The transformation implementation should be considered as a possible area for follow-on thesis work.

Another area which has not received much attention in this thesis is the design and implementation of the end user's interface. This interface should be considerably more structured than that of the previously cited programmer's interface. The end user should be able to query the database through a sequence of menu calls or icon selections, rather than be required to learn and to use a structured query language such as SQL or EQUOL for the same results. And finally, the end user's interface for sound manipulations must be equally compatible with the other forms of multimedia information within the integrated database.

LIST OF REFERENCES

1. Meyer-Wegener, K., V. Lum and C. Wu, *Image Database Management in a Multimedia System*, Naval Postgraduate School, Monterey, California, April 1988.
2. *MacRecorder: Introduction To Sound*, Farallon Computing, Inc., Berkeley, California, pp. 2-17, 1987.
3. Tanenbaum, Andrew S., *Computer Networks*, Prentice Hall, Inc., Englewood Cliffs, New Jersey, pp. 104-110, 1981.
4. Crochiere, Ronald E. et al., "Real-Time Speech Coding," *IEEE Transactions on Communications*, Vol. COM-30, No. 4, pp. 621-631, April 1982.
5. Frantz, Gene and K. Ling, "Speech Technology in Consumer Products," *Speech Technology*, Vol. 1, No. 2, pp. 25-34, April 1982.
6. Myers, Andrew B. (ed.), "Speech Processing Technology," *AT&T Technical Journal*, Vol. 65, Issue 5, September/October 1986.
7. Bertino, Elisa et al., "Query Processing In A Multimedia Document System," *ACM Transactions On Office Information Systems*, Vol. 6, No.1, pp. 1-41, January 1988.
8. Sanders, Mark S. and E. McCormick, *Human Factors In Engineering and Design*, McGraw-Hill Book Company, New York, 6th ed., pp. 140-147, 1987.
9. Steinbrecher, David, "Optical Disks Go Head To Head With Traditional Storage Media," *Today's Office*, pp. 24-30, October 1987.
10. Terry, Douglas B. and D. Swinhart, "Managing Stored Voice In The Etherphone System," *ACM Transactions on Computer Systems*, Vol. 6, No. 1, pp. 3-27, February 1988.
11. Lockemann, Peter C., *Multimedia Databases: A Paradigm and an Architecture*, Naval Postgraduate School, Monterey, California, August 1988.
12. Christodoulakis, S. et al., "Multimedia Document Presentation, Information Extraction, and Document Formation In MINOS: A Model And A System," *ACM Transactions On Office Information Systems*, Vol.4, No. 4, pp. 345-383, October 1986.
13. Gould, J. D. and S. J. Boies, "Speech Filing--An Office System for Principles," *IBM Systems Journal*, Vol. 23, No. 1, pp. 65-81, January 1984.

14. Postel, Jonathan B. et al., "An Experimental Multimedia Mail System," *ACM Transactions On Office Information Systems*, Vol. 6, No. 1, pp. 63-81, January 1988.
15. Thomas, R. H. et al., "DIAMOND: A Multimedia Message System Built on a Distributed Architecture," *Computer*, Vol. 18, No. 12, pp. 65-78, December 1985.
16. Maxemchuk, N., "An Experimental Speech Storage and Editing Facility," *Bell Systems Technical Journal*, Vol. 59, pp. 1383- 1395, 1980.
17. Meyer-Wegener, K., *A Project on Multimedia Databases*, Naval Postgraduate School, Monterey, California, April 1988.
18. Meyer-Wegener, K., *Extending an SQL Interface with the Data Type IMAGE*, Naval Postgraduate School, Monterey, California, July 1988.

APPENDIX A - THE SQL PREPROCESSOR OVERVIEW

The abstract data type SOUND is not recognizable to standard INGRES SQL. So every occurrence of a SOUND attribute must first be translated by the preprocessor into the internal representation of the structure, i.e. a pair of declarations that are recognizable by standard INGRES SQL. In this case, the declarations could correspond to a pair of program variables consisting of the filename of the SOUND object and the description of the contents of this object. The resulting input query language used by the preprocessor is referred to as SOUND SQL (SSQL).

The preprocessor reads the source file, looks for SSQL statements, checks whether they affect SOUND attributes, and replaces them. While doing so, it collects information that has to be included into the source. (e.g., the declaration of additional variables). To identify the associated filename attribute, a "_f" extension is attached to the name of the SOUND attribute. For the description attribute identification, a "_d" extension is attached. INGRES Embedded SQL uses the "exec sql" syntax to initiate the SQL statements. A demonstration of the use of embedded SQL in conjunction with the internal and external sound related functions are described below. See Chapter V for a review of these functions. The "SS" prefix is used to denote internal functions and SOUND-SQL variables.

The following discussions concerning the transformation of SQL statements follows both the SQL Quick Reference Summary and [Ref. 18].

* * * * *

The **UPDATE** operation:

(user input)

```
exec sql UPDATE SOUND_OBJECT
      SET S_SOUND = CONSTRUCT_SOUND(size, duration,...)
      WHERE S_ID = 292;
```

(preprocessor transformations)

```
exec sql begin declare section;
      char SSfilename1[64];
      char SSdescr1[500];
exec sql end declare section;
SSconstruct_sound ( size, duration, ..., SSfilename1,
                  SSdescr1);
```

The bracketed "exec sql" sequence of statements declare and identify SOUND-SQL (SS) variables. The internal function "SSconstruct_sound" behaves as a procedure and takes as input the same parameters presented to the external CONSTRUCT_SOUND function, and returns as output the filename (SSfilename1) and description (SSdescr1) of the SOUND object. The return value of the internal functions are used to pass error codes or status information.

The return parameters of the internal functions are then used for the translation of a SSQL statement into its INGRES Embedded SQL equivalent. This, in effect, removes the SSQL statements from the resulting program. The final conversion of the UPDATE statement appears below.

```

exec sql UPDATE SOUND_OBJECT
      SET      S_SOUND_F = :SSfilename1;
            S_SOUND_D = :SSdescr1;
      WHERE    S_ID = 292;

```

Other functions can be similarly described. A few of these are listed below.

INSERT Example:

```

exec sql INSERT INTO SOUND-OBJECT
            (S_ID, S_SOUND)
      VALUES (3212, CONSTRUCT_SOUND(filename,...) );

```

(preprocessor transformations)

```

exec sql begin declare section;
      char SSfilename1[64];
      char SSdescr1[500];
exec sql end declare section;
SSconstruct_sound ( size, duration, ..., SSfilename1,
                  SSdescr1);

exec sql INSERT INTO SOUND_OBJECT
      (S_ID, S_SOUND_F, S_SOUND_D)
      VALUES (312, :SSfilename1, :SSdescr1);

```

ADD_DESCRIPTION Example:

```
exec sql UPDATE SOUND_OBJECT
SET S_SOUND = ADD_DESCRIPTION(S_SOUND,new_descr)
WHERE S_ID = 292;
```

(preprocessor transformations)

```
exec sql begin declare section;
      char SSfilename2[64],
           SSfilename3[64];
      char SSdescr2[500]
           SSdescr3[500];
exec sql end declare section;
exec sql SELECT S_SOUND_F, S_SOUND_D
      INTO   :SSfilename2, :SSdescr2
      FROM   SOUND_OBJECT
      WHERE  S_ID = 292;
SSadd_description (SSfilename2, SSdescr2, new_descr,
      SSfilename3, SSdescr3);
exec sql UPDATE SOUND_OBJECT
      SET S_SOUND_F = :SSfilename3,
      S_SOUND_D = :SSdescr3;
      WHERE S_ID = 292;
```

PLAY_SOUND Example:

```
exec sql SELECT  PLAY_SOUND(S_SOUND),
                DESCRIPTION(S_SOUND)
        INTO   :var1, :var2
        FROM   SOUND_OBJECT
        WHERE  IS_INCLUDED_IN(S_SOUND, "grey whale");
```

(preprocessor transformations)

```
exec sql begin declare section;
        char SSfilename4[64];
        char SSdescr4[500];
        int  SSfound;
exec sql end declare section;
exec sql declare SSc1 cursor for
        SELECT S_SOUND_F, S_SOUND_D
        FROM   SOUND_OBJECT
exec sql open SSc1 cursor;
exec sql whenever not found goto SScloseSSc1;
for (;;)
{
        exec sql FETCH SSc1
                INTO :SSfilename4, :SSdescr4;
        SSis_included_in(SSfilename4, SSdescr4,
                        "greywhale", SSfound);
```

```

        if (SSfound = TRUE)
        {
            SSplay_sound(SSfilename4, SSdescr4, &var1);
            SSdescription(SSfilename4, SSdescr4, &var2);
            goto SScloseSSc1;
        }
    }

SScloseSSc1:
    exec sql close SSsc1;
exec sql whenever not found "old action";

```

The preprocessor is not required to generate exactly this code, but the effect should be the same.

APPENDIX B - THE INTERNAL SOUND HANDLER FUNCTIONS

/* **** */
/* **** */

AUTHOR: SAWYER, GREGORY R.
RANK: LCDR, USN
ADVISOR: PROF VINCENT Y. LUM
CO-ADVISOR: ADJ PROF KLAUS MEYER_WEGENER

THESIS TITLE: MANAGING SOUND IN A
RELATIONAL MULTIMEDIA
DATABASE SYSTEM

GRAD. DATE: 15 DECEMBER 1988

Submitted as partial fulfillment of a thesis requirement relative to the receipt of a Masters of Science Degree in Computer Science from the Naval Postgraduate School at Monterey, California.

/* **** */
/* **** */

This is a sound handling module for a multimedia information system which can be supported by a variety of multimedia data bases using standard "C" and an Antex VP620E sound board. The functions can be called from almost any program. The functions included in this module directly support the software driver of the VP620E. Modifications to this module to support other drivers would require replacing the function calls to the VP620E software driver with those which support the new driver.

An automatic header for each file which includes the size, sampling rate, encoding, duration and resolution is at the start of each recorded file. The following functions are the operations necessary to realize the full power of the abstract data type known as SOUND.

THE NAME OF THIS FILE IS: SND_STRU.C

```
/* **** */
/* **** */
```

```
/* This structure represents the sound object whose features
   will be stored in the file as a header prefix to each
   recorded file. The database information will consist only
   of the unique file identifier and the description data.
*/
```

```
struct SND_HDR
{
    long int s_size;           /* number of bytes */
    int s_samplerate;         /* 8K or 16K per sec */
    int s_encoding;           /* 0=none, 1=ADPCM */
    float s_duration;         /* time in sec and hundredths */
    int s_resolution;         /* # bits per sample */
}hdr_info;
```

THE NAME OF THIS FILE IS : SND_ERRS.C

```
/* **** */
/* **** */
```

```
/* This module contains a list of possible I/O error responses.
```

```
   This list is truly extensible.
```

```
*/
```

```
typedef enum
```

```
{
    PARS, WOPEN, WRITE, WCLOSE, ROPEN, READ, RCLOSE, SRATE,
    TOO_LONG, OK
} ERROR;
```

```
void displayerr(e)
```

```
ERROR e;
```

```
{
```

```
    switch (e)
```

```
    {
```

```
        case PARS      : printf("Incorrect parameters\n");
                          return;
```

```
        case WOPEN     : printf("Cannot open file for output\n");
                          return;
```

```
        case WRITE     : printf("File write error\n");
                          return;
```

```
        case WCLOSE    : printf("Cannot close output file\n");
                          return;
```

```
        case ROPEN     : printf("Cannot open file for input\n");
                          return;
```

```
        case READ      : printf("File read error\n");
                          return;
```

```
        case RCLOSE    : printf("Cannot close input file\n");
                          return;
```

```
        case SRATE     : printf("Incompatible sampling rates for files\n");
                          return;
```

```
        case TOO_LONG  : printf("Description is too long\n");
                          return;
```

```
    }
```

```
}
```

THE NAME OF THIS FILE IS: SF_NAME.C

```
/* **** */
/* **** */
/*
**** This module converts the standard time parameters of GMT
      into 1-digit hexadecimal numbers which enables the
      construction of a unique filename for a sound file.
*/
```

```
char YR(yr)
int yr;
{
    switch(yr)
    {
        case 88: return '8'; break;
        case 89: return '9'; break;
        case 90: return '0'; break;
        case 91: return '1'; break;
        case 92: return '2'; break;
        case 93: return '3'; break;
        case 94: return '4'; break;
        case 95: return '5'; break;
        case 96: return '6'; break;
        case 97: return '7'; break;
        case 99: return 'a'; break;
    }
}
```

```
/* ----- */
```

```
char MN(mn)
int mn;
{
    switch (mn)
    {
        case 1: return '1'; break;
        case 2: return '2'; break;
        case 3: return '3'; break;
        case 4: return '4'; break;
        case 5: return '5'; break;
        case 6: return '6'; break;
        case 7: return '7'; break;
        case 8: return '8'; break;
        case 9: return '9'; break;
        case 10: return 'A'; break;
        case 11: return 'B'; break;
        case 12: return 'C'; break;
    }
}
```

```
/* ----- */
```

```
char DAY(day)
int day;
{
    switch (day)
    {
        case 1: return '1'; break;
        case 2: return '2'; break;
        case 3: return '3'; break;
        case 4: return '4'; break;
        case 5: return '5'; break;
        case 6: return '6'; break;
        case 7: return '7'; break;
        case 8: return '8'; break;
        case 9: return '9'; break;
        case 10: return 'a'; break;
        case 11: return 'b'; break;
        case 12: return 'c'; break;
        case 13: return 'd'; break;
        case 14: return 'e'; break;
        case 15: return 'f'; break;
        case 16: return 'g'; break;
        case 17: return 'h'; break;
        case 18: return 'i'; break;
        case 19: return 'j'; break;
        case 20: return 'k'; break;
        case 21: return 'l'; break;
        case 22: return 'm'; break;
        case 23: return 'n'; break;
        case 24: return 'o'; break;
        case 25: return 'p'; break;
        case 26: return 'q'; break;
        case 27: return 'r'; break;
        case 28: return 's'; break;
        case 29: return 't'; break;
        case 30: return 'u'; break;
        case 31: return 'v'; break;
    }
}
```

```
/* ----- */
```

```
char HR(hr)
```

```
int hr;
```

```
{
```

```
    switch (hr)
```

```
    {
```

```
        case 1: return '1'; break;
```

```
        case 2: return '2'; break;
```

```
        case 3: return '3'; break;
```

```
        case 4: return '4'; break;
```

```
        case 5: return '5'; break;
```

```
        case 6: return '6'; break;
```

```
        case 7: return '7'; break;
```

```
        case 8: return '8'; break;
```

```
        case 9: return '9'; break;
```

```
        case 10: return 'a'; break;
```

```
        case 11: return 'b'; break;
```

```
        case 12: return 'c'; break;
```

```
        case 13: return 'd'; break;
```

```
        case 14: return 'e'; break;
```

```
        case 15: return 'f'; break;
```

```
        case 16: return 'g'; break;
```

```
        case 17: return 'h'; break;
```

```
        case 18: return 'i'; break;
```

```
        case 19: return 'j'; break;
```

```
        case 20: return 'k'; break;
```

```
        case 21: return 'l'; break;
```

```
        case 22: return 'm'; break;
```

```
        case 23: return 'n'; break;
```

```
        case 24: return 'o'; break;
```

```
    }
```

```
}
```

```
/* ----- */
```

```

/*****
/*****
/*
    AUTHOR:      SAWYER, GREGORY R.
    RANK:        LCDR, USN
    ADVISOR:     PROF. VINCENT Y. LUM
    CO-ADVISOR:  ADJ. PROF. KLAUS MEYER_WEGENER

    THESIS TITLE:  MANAGING SOUND IN A
                   RELATIONAL MULTIMEDIA
                   DATABASE SYSTEM

    GRAD. DATE:   15 DECEMBER 1988
*/
/*****
/*****
/*****

    THE NAME OF THIS FILE IS: SND_FNCS.C

/*****
/*****

#include <stdio.h>
#include <sys/types.h>
#include <time.h>
#include "snd_stru.c"
#include "snd_errs.c"
#include "sf_name.c"

#define NAME_LENGTH 12
#define MAX_DESCR 500
#define ERROR_FREE 0
#define SOUND_ERROR -1

#define BEGIN 1;
#define SETREC 2;
#define START 4;
#define STOP 5;
#define STATUS 6;
#define PLAY 8;
#define END 9;

int VP620 ();

/*****
/*****

```

```

/* ----- */
/* *** ADD DESCRIPTION *** */
/* ----- */

SSadd_description(infile, indscr, newdescr, outfile, outdescr)

/*
**** add to the description of a sound object
*/

char *infile, /* input */
      *indscr, /* input */
      *newdescr, /* input */
      *outfile, /* output */
      *outdescr; /* output */

{
    int i = 0;

    while (*outfile++ = *infile++)
        ;

    while (*outdescr++ = *indscr++)
        i++;

    outdescr--; /* reposition on '\0' */

    while ((*outdescr++ = *newdescr++) && i < MAX_DESCR)
        i++;

    if (i == MAX_DESCR && *outdescr != '\0')
    {
        *outdescr = '\0';
        displayerr(TOO_LONG);
        return SOUND_ERROR;
    }
    return ERROR_FREE;
}

```

```

/* ----- */
/* *** REPLACE DESCRIPTION *** */
/* ----- */

SSreplace_description(infile, indescr, newdescr, outfile, outdescr)

/*
**** replace the description of a sound object
*/

char *infile,          /* input */
    *indescr,          /* input */
    *newdescr,         /* input */
    *outfile,          /* output */
    *outdescr;         /* output */

{
    while (*outfile++ = *infile++)
        ;
    while (*outdescr++ = *newdescr++)
        ;
    return ERROR_FREE;
}

```

```

/* ----- */
/* *** DESCRIPTION LENGTH *** */
/* ----- */

```

SSdescription_length (filename, descr, char_count)

```

/*
**** count the characters in the description of a sound object
*/

```

```

char *filename;          /* input, not used */
char *descr;             /* input */
unsigned int *char_count; /* output */

```

```

{
    unsigned int i = 0;

    while (*descr++ != '\0')
        i++;

    *char_count = i;
    return ERROR_FREE;
}

```

```

/* ----- */
/* *** DESCRIPTION *** */
/* ----- */

```

SSdescription (filename, old_descr_name, new_descr_name)

```

/*
**** copy the description to the output
*/

```

```

char *filename,          /* input, not used */
    *old_descr_name,     /* input */
    *new_descr_name;     /* output */

{
    while (*new_descr_name++ = *old_descr_name++)
        ;
    return ERROR_FREE;
}

```

```

/* ----- */
/* *** IS_INCLUDED_IN *** */
/* ----- */

```

SSis_included_in (filename, descr, pattern, match)

```

/*
**** determine whether or not string "pattern" is contained
**** within string "descr"; returns 1 if true, 0 if false
*/

char *filename,      /* input, not used */
    *descr,          /* input */
    *pattern;        /* input */
int *match;          /* output */

{
    int i,
        j,
        found;

    if (*pattern == '\0')
        found = 1; /* NULL string always is contained */
    else
        found = 0; /* initialize found for loop use */

    i = 0;

    while (*(descr+i) != '\0' && !found)
    {
        if (*(descr+i) == *pattern)
        {
            j = 0;
            while (*(descr+i+j) == *(pattern+j) && *(pattern+j) != '\0')
                j++;
            if (*(pattern+j) == '\0') /* pattern matched */
                found = 1;
            else
                if (*(descr+i+j) == '\0') /* pattern longer than
                                           remaining descr */
                    i = i + j; /* terminate outer loop */
                else /* continue search starting with next letter in descr */
                    i++;
        }
        else
            i++;
    }

    *match = found;
    return ERROR_FREE;
}

```

```

/* ----- */
/* *** SIZE *** */
/* ----- */

```

SSsize(fname,descr,size)

```

/*
**This function reads a header from a designated file,
returns the updated header, then passes the size
as a long int back to the caller. No Error Checking Performed.
*/

```

```

char *fname;          /* input */
char *descr;          /* not used */
long int size;        /* return value */
{
    struct SND_HDR hdr;

    read_snd_hdr(fname,&hdr);    /* read hdr fields */

    size = hdr.s_size;
    return ERROR_FREE;
}

```

```

/* ----- */
/* *** SAMPLING RATE *** */
/* ----- */

```

SSsamplerate(fname,descr,samprate)

```

/*
***** This function reads a header from a designated file,
returns the updated header, then passes the samplerate
as an int back to the caller. No Error Checking Performed!
*/

```

```

char *fname;          /* input */
char *descr;          /* not used */
int samprate;         /* return value */
{
    struct SND_HDR hdr;

    read_snd_hdr(fname,&hdr);    /* read hdr fields */

    samprate = hdr.s_samplerate;
    return ERROR_FREE;
}

```

```

/* ----- */
/* *** ENCODING *** */
/* ----- */

SSencoding(fname,descr,encoding)
/*
***** This function reads a header from a designated file,
        returns the updated header, then passes the encoding code
        as an int back to the caller. No Error Checking Performed!
*/

char *fname;          /* input */
char *descr;          /* not used */
int encoding;         /* return value */
{
    struct SND_HDR hdr;

    read_snd_hdr(fname,&hdr);    /* read hdr fields */

    encoding = hdr.s_encoding;
    return ERROR_FREE;
}

/* ----- */
/* *** DURATION *** */
/* ----- */

SSduration(fname,descr,duration)
/*
***** This function reads a header from a designated file,
        returns the updated header, then passes the duration in
        seconds and hundredths of a second as a float back to the
        caller. No Error Checking Performed!
*/

char *fname;          /* input */
char *descr;          /* not used */
float duration;       /* return value */
{
    struct SND_HDR hdr;

    read_snd_hdr(fname,&hdr);    /* read hdr fields */

    duration = hdr.s_duration;
    return ERROR_FREE;
}

```

```

/* ----- */
/* *** RESOLUTION *** */
/* ----- */

SSresolution(fname,descr,resolution)
/*
***** This function reads a header from a designated file,
        returns the updated header, then passes the resolution in
        in number of bits-per-sample as an int back to the
        caller. Error Checking Performed to determine if the hdr field contains
        garbage!
*/

char *fname;           /* input */
char *descr;           /* not used */
int resolution;        /* return value */
{
    struct SND_HDR hdr;

    read_snd_hdr(fname,&hdr);    /* read hdr fields */

    /* check to see if this field contains garbage */
    if ( (hdr.s_resolution > 32) || (hdr.s_encoding != 1) )
    {
        hdr.s_resolution = 0;
        return SOUND_ERROR;
    }

    resolution = hdr.s_resolution;
    return ERROR_FREE
}

```

```

/* ----- */
/* *** READ SOUND HEADER *** */
/* ----- */

read_snd_hdr(fname,h)
/*
***** This function reads a header from a designated file,
        and returns the header to the caller with the various
        fields updated.
*/

char  *fname;                /* output */
struct SND_HDR  *h;          /* sound object record */
{
    FILE *f;
    int  num;

    if ((f = fopen(fname,"rb")) == NULL)    /* open for reading */
    {
        displayerr(ROPEN);
        return(SOUND_ERROR);
    }

    num = 1;                            /* only one header */

    /* ***** read the header from the predesignated input file */
    if (fread(h, sizeof(struct SND_HDR), 1, f) < num )
    {
        displayerr(READ);                /* read error */
        return(SOUND_ERROR);
    }

    if (fclose(f) != 0)
    {
        displayerr(WCLOSE);              /* close error */
        return(SOUND_ERROR);
    }

    return(0);
}

```

```

/* ----- */
/* *** CONCATENATE_SOUNDS *** */
/* ----- */

SSconcatenate_sounds(fname1,fname2,newfile)
/*
***** This function concatenates two sound files and generates
        a new file as a result.
        This function should return a "0" if successful.

*/

char *fname1;          /* input file #1 */
char *fname2;          /* input file #2 */
char *newfile;         /* output file */
{
    struct SND_HDR hdr1,hdr2,hdr3;
    FILE *f,*fg,*fh;
    char *buf[500];     /* input/out buffer */
    int num = 1;        /* only one header */

    read_snd_hdr(fname1,&hdr1);
    read_snd_hdr(fname2,&hdr2);

    if (hdr1.s_samplerate == hdr2.s_samplerate)
    {
        hdr3.s_size = hdr1.s_size + hdr2.s_size;
        hdr3.s_samplerate = hdr1.s_samplerate;
        hdr3.s_encoding = hdr1.s_encoding;
        hdr3.s_duration = hdr1.s_duration + hdr2.s_duration;
        hdr3.s_resolution = hdr1.s_resolution;

        generate_filename(newfile);

        if ((f = fopen(newfile,"wb")) == NULL) /* open for writing */
        {
            displayerr(WOPEN);
            return SOUND_ERROR;
        }

        if ((fg = fopen(fname1,"rb")) == NULL) /* open for reading */
        {
            displayerr(ROPEN);
            return SOUND_ERROR;
        }

        if ((fh = fopen(fname2,"rb")) == NULL) /* open for reading */
        {
            displayerr(ROPEN);
            return SOUND_ERROR;
        }
    }
}

```

```

}

/* ***** write the header into the predesignated output file */
if (fwrite(&hdr3, sizeof(struct SND_HDR), 1, f) < num )
{
displayerr(WRITE);          /* write error */
return SOUND_ERROR;
}

while (!feof(fg))
{
if (fread(buf,500,1,fg) < 0 )      /* load buffer */
{
displayerr(READ);
return SOUND_ERROR;
}

/* ***** append data from sound data buffer */
if (fwrite(buf, 500, 1, f) < num )      /* write buffer */
{
displayerr(WRITE);
return SOUND_ERROR;
}
}

while (!feof(fh))
{
if (fread(buf,500,1,fh) < 0 )      /* load buffer */
{
displayerr(READ);
return SOUND_ERROR;
}

/* ***** append data from sound data buffer */
if (fwrite(buf, 500, 1, f) < num )      /* write buffer */
{
displayerr(WRITE);
return SOUND_ERROR;
}
}

printf("Data successfully written...\n");

if ((fclose(f) != 0) && (fclose(fg) != 0) && (fclose(fh) != 0))
{
displayerr(WCLOSE);          /* close error */
return SOUND_ERROR;
}

```

```

    return ERROR_FREE;
}
else
{
    displayerr(SRATE);
    return SOUND_ERROR;
}
}

/* ----- */
/* *** PLAY SEGMENT *** */
/* ----- */

SSplay_segment(fname,start_time,stop_time)
/*
***** This function plays only the designated portion of a file
        which the user desires vice the entire sound data file.
        This function should return a "0" if successful.

*/

char *fname;                /* input file */
float start_time;
float stop_time;
{
    char *p;
    char *file_segment;      /* combined file */

    /* concatenate 'fname' and 'times' to form a single
       filename for use by the 'play' function
    */

    *file_segment = malloc(30);
    sprintf(file_segment, "%s/%07.2f/%07.2f",fname,
            start_time,stop_time);

    file_segment[26] = '\0';

    for (p=file_segment; *p; p++)
        if (*p==' ')
            *p = 0;

    if (SSantex_play(fname,file_segment) != 0)
        return SOUND_ERROR;    /* oops... an error */
    else
        return ERROR_FREE;
}

```

```

/* ----- */
/* *** LINK SOUNDS *** */
/* ----- */

SSlink_sounds(fname1,fname2)
/*
***** This function plays two sound object files back-to-back.
        This function should return a "0" if successful.
*/

char *fname1;
char *fname2;
{

    char *p;
    char *file_segment;

    *file_segment = malloc(30);
    sprintf(file_segment,"%s,%s",fname1,
            fname2);

    file_segment[25] = '\0';

    for (p=file_segment; *p; p++)
        if (*p==' ')
            *p = 0;

    SSantex_play(fname1,file_segment);

    return ERROR_FREE;
}

```

```

/* ----- */
/* *** CUT_SEGMENT *** */
/* ----- */

SScut_segment(fname,time1,time2,newfile)
/*
   This function cuts a designated segment out of the
   specified file and creates a new file minus the segment.
*/

char *fname;           /* input */
float time1;           /* segment start */
float time2;           /* segment end */
char *newfile;         /* output */
{
    FILE *f,*fg,
    char *buf[200];
    int temp = 0;
    int lower_bound = 0;
    int upper_bound = 0;
    int num = 1;
    struct SND_HDR h,r;

    read_snd_hdr(fname,&h);    /* get header info of input file */

    generate_filename(newfile); /* create a new output file */

    /* compute the segment time boundaries */
    if (h.s_samprate == 8000)
    {
        lower_bound = 4000 * time1;
        upper_bound = 4000 * time2;
    }
    else
    {
        lower_bound = 8000 * time1;
        upper_bound = 8000 * time2;
    }

    /* *** update header for new output file */
    r.s_size = h.s_size - (upper_bound - lower_bound);
    r.s_samprate = h.s_samprate;
    r.s_encoding = h.s_encoding;
    r.s_duration = h.s_duration - (time2 - time1);
    r.s_resolution = h.s_resolution;

    if ((f = fopen(newfile,"wb")) == NULL ) /* open for writing */
    {
        displayerr(WOPEN);
        return SOUND_ERROR;
    }
}

```

```

    }

    if ((fg = fopen(fname, "rb")) == NULL) /* open for reading */
    {
        displayerr(ROPEN);
        return SOUND_ERROR;
    }

    /* *** write the header into the predesignated output file */
    if (fwrite(&r, sizeof(struct SND_HDR), 1, f) < num )
    {
        displayerr(WRITE);
        return SOUND_ERROR;
    }

    /* *** now load the buffer and write to the output file */
    while (!feof(fg))
    {
        if ( (fread(buf,200,1,fg)) >= 0 )
        {
            temp = temp + 200;

            if ( (temp >= lower_bound) || (temp <= upper_bound) )
                fwrite(buf,200,1,f); /* hmm...no error checking */
        }
        else
        {
            displayerr(READ);
            return SOUND_ERROR;
        }
    }
    return ERROR_FREE;
}

```

```

/* **** */

/*      ***** ANTEX_PLAY FUNCTION      **** */

/* **** */
SSantex_play(filename,temp_fname)
/*
    This is the actual function that plays the sound. Its input is
    two filenames, the second of which may not be used. A successful
    play will return a '0' to the caller. Failure will return an
    error message.
*/

char *filename;                /* primary input file */
char *temp_fname;             /* combined file */
{
    /* declarations */
    int port,useint;
    int vpfunction,samplerate;
    int state,error,sec,hundsec,overload;

    int monitor = 1;          /* record monitor always on */

    long int sz;
    int srate,sencode,sresol;
    float sdur;

    ERROR err;
    struct SND_HDR hdr;

    /* ***** Executable Statements ***** */

    if ((read_snd_hdr(filename,&hdr)) == 0)
    {
        if ((strpbrk(temp_fname,".snd")) == NULL)
            strcpy(temp_fname, filename);

        /* ***** read header values to set parameters */
        printf("Current File = %s\n",temp_fname);
        printf("Size=%ld Srate=%d Enc=%d Dur=%5f Resol=%d\n",
            hdr.s_size,hdr.s_samplerate,hdr.s_encoding,
            hdr.s_duration,hdr.s_resolution);

        vpfunction = BEGIN;    /* alert to the driver */

        port = 0x280;
        useint = 2;
        VP620 (&vpfunction, &useint, &port);
    }
}

```

```

if (hdr.s_samplerate == 8000)
    srate = 0;
else
    srate = 1;

vpfunction = PLAY;
VP620 (&vpfunction, &srate, temp_fname);    /* open file */

vpfunction = STATUS;

do {
    VP620 (&vpfunction, &overload, &hundsec, &sec, &error, &state);
    printf(" State= %d Error = %d Sec = %d.%d Overload = %d \r",
        state, error, sec, hundsec, overload);
}
while (!kbhit() & state != 3);

printf("\n");
printf("End of play! \n");

/* these statements always required to close the file */
vpfunction = STOP;
VP620 (&vpfunction);

vpfunction = END;
VP620 (&vpfunction);

return ERROR_FREE;
}
else
{
    displayerr(READ);
    return SOUND_ERROR;
}
}

```

```

/* ----- */
/* *** Generate A New File Name *** */
/* ----- */
generate_filename(sound_filename)
/*
***** Produce a unique 8-digit filename for the recording composed of 1-digit each for
year, month & hour, and 2-digits each for minute and second. Each sound object
can be identified by the ".snd" suffix.
This code is similar to that used by the IMAGE functions.
*/

char *sound_filename;
{
    char *p;
    struct tm *t;
    time_t current_time;

    current_time = time(NULL);

    t = gmtime(&current_time);

    sprintf(sound_filename, "%1c%1c%1c%1c%2d%2d.%s",
        YR(t->tm_year), MN(t->tm_mon), DAY(t->tm_mday),
        HR(t->tm_hour),
        t->tm_min, t->tm_sec, "snd");

    sound_filename[NAME_LENGTH-1] = '\0';

    for (p=sound_filename; *p; p++)
        if (*p == ' ')
            *p = '0';

    return *sound_filename;
}

```

```

/* ----- */
/* *** Store Sound Header and Data *** */
/* ----- */

ERROR store_snd_hdr(fname,r,temp_file)
/*
***** This function stores a header into a designated file,
        then reads the recorded sound file, buffers the data,
        then writes the buffer into the designated file following
        the header.
*/

char  *fname;                /* output */
struct SND_HDR r;            /* sound object record */
char  *temp_file;
{
    FILE *f,*fg;
    char *buf[500];           /* input/out buffer */
    int  num = 1;              /* only one header */

    if ((f = fopen(fname,"wb")) == NULL)    /* open for writing */
        return(WOPEN);

    if ((fg = fopen(temp_file,"rb")) == NULL) /* open for reading */
        return(ROPEN);

    /* ***** write the header into the predesignated output file */
    if (fwrite(&r, sizeof(struct SND_HDR), 1, f) < num )
        return (WRITE);          /* write error */

    while (!feof(fg))
    {
        if (fread(buf,500,1,fg) < 0 )          /* load buffer */
            return(READ);

        /* ***** append data from sound data buffer */
        if (fwrite(buf, 500, 1, f) < num )      /* write buffer */
            return (WRITE);
    }

    if (fclose(f) != 0)
        return(WCLOSE);                /* close error */

    if (fclose(fg) != 0)
        return(WCLOSE);
}

```

```

    return(OK);
}

/* ----- */
/* *** Determine Size of DATA-ONLY File *** */
/* ----- */

long FileSize(i_file)
char *i_file;                /* input file */
{
    FILE *f;
    long int f_size;

    if ((f = fopen(i_file, "rb")) == NULL) /* open file */
        displayerr(ROPEN);

    if (fseek(f,0L,2) != 0)                /* set position rel. to end */
        return EOF;

    f_size = ftell(f);

    if (fclose(f) != 0)                    /* close file */
        displayerr(RCLOSE);

    return f_size;
}

```

```

/* **** */

/*          ANTEX_RECORD FUNCTION          */

/* **** */
SSantex_record(filename)
char *filename;
{
    int port,useint;
    int vpfunction,samplerate;
    int state,error,sec,hundsec,overload;
    int monitor = 1;          /* record monitor always on */

    long int sz;
    int srate,sencode,sresol;
    float sdur;
    int c;

    int *pi,                  /* storage allocation stmts...unused */
        i = 0;
    char *newname;

    ERROR err;
    struct SND_HDR hdr;

    /* ***** Executable statements ***** */

    generate_filename(filename);

    vpfunction = BEGIN;          /* vpbegin */
    port = 0x280;                /* use default IO address */
    useint = 2;                  /* use interrupt 2 */
    VP620(&vpfunction,&useint,&port); /* wake-up call to driver */

    vpfunction = SETREC;          /* vpsetrec */
    samplerate = 0;
    sencode = 1;                 /* ANTEX recording w/8bit resolution */

    newname = "temp.snd";
    VP620(&vpfunction,&monitor,&samplerate,newname);

    puts("Press ENTER to begin... \n");
    c = getchar();

    vpfunction = START;          /* vpstart */
    VP620(&vpfunction);

    printf("Recording in progress...Press any key to stop!\n");

```

```

vpfunction = STATUS;                                /* vpstatus */
do
{
    VP620(&vpfunction,&overload,&hundsec,&sec,&error,&state);
    printf(" State=%d Error=%d Seconds=%d.%02d Overload=%d\n",
        state,error,sec,hundsec,overload);

}
while(!kbhit() & state!=3);

/* These statements always required to close the file */
vpfunction = STOP;                                  /* vpstop */
VP620(&vpfunction);

printf("\n");
printf("End of recording session! \n");

vpfunction = END;                                    /* vpend */
VP620(&vpfunction);

/* Update the header fields now that the file has been recorded */
sz = FileSize(newname) + sizeof(struct SND_HDR);
hdr.s_size = sz;

if (samplerate == 0)
    srate = 8000;
else
    srate = 16000;

hdr.s_samplerate = srate;
hdr.s_encoding = sencode;                            /* ADPCM code */

sdur = (sec + ((float) hundsec / 100));
hdr.s_duration = sdur;

if (sencode == 1)
    hdr.s_resolution = 8;                            /* set #bits-per-sec */
else
    hdr.s_resolution = 0;

printf("Header info--> size=%ld srate=%d enc=%d dur=%5f resol=%d\n",
    hdr.s_size,hdr.s_samplerate,hdr.s_encoding,
    hdr.s_duration, hdr.s_resolution);

/* ***** store header and data into designated file */
if ((err = store_snd_hdr(filename,hdr,newname)) != OK)
    displayerr(err);
}
/* ***** */
/* ***** */

```

BIBLIOGRAPHY

Atal, Bishnus S., "Predictive Coding Of Speech At Low Bit Rates," *IEEE Transactions on Communications*, Vol. COM-30, No. 4, April 1982.

Badgett, T., "Searching Through Files with Database Software," *PC Magazine*, Vol. 6, No. 18, October 1987.

Bertino, Elisa et al., "Query Processing In A Multimedia Document System," *ACM Transactions On Office Information Systems*, Vol. 6, No.1, January 1988.

Christodoulakis, S. et al., "Multimedia Document Presentation, Information Extraction, and Document Formation In MINOS: A Model And A System," *ACM Transactions On Office Information Systems*, Vol.4, No. 4, October 1986.

Crochiere, Ronald E. et al., "Real-Time Speech Coding," *IEEE Transactions on Communications*, Vol. COM-30, No. 4, April 1982.

Demurjian, S. A., and Hsiao, D. K., "The Multi-Lingual Database System," *Proceedings of the Third International Conference on Data Engineering*, Los Angeles, California, February 1987.

Demurjian, S. A., and Hsiao, D. K., *Towards a Better Understanding of Data Models Through the Multi-Lingual Database System*, NPS52-87-018, Naval Postgraduate School, Monterey, California, May 1987.

Dickey, Sam, "CD-ROM: A World Of Information On A Disk," *Today's Office*, June 1987.

Flanagan, James L. et al., "Digital Voice Storage In A Microprocessor," *IEEE Transactions on Communications*, Vol. COM-30, No. 2, February 1982.

Fossum, Robert R. and V. Cerf, "Communications Challenges For The 80's," *SIGNAL*, October 1979.

Frantz, Gene and K. Ling, "Speech Technology in Consumer Products," *Speech Technology*, Vol. 1, No. 2, April 1982.

Georgiou, Bill, "Give An Ear To Your Computer: A Speech Recognition Primer For Computer Experimenters," *BYTE*, BYTE Publications, Inc., June 1978.

Gibbs, Simon et al., "MUSE: A Multimedia Filing System," *Computer Society of the IEEE*, Vol. 4, No. 2, March 1987.

Gould, J. D. and S. J. Boies, "Speech Filing--An Office System for Principles," *IBM Systems Journal*, Vol. 23, No. 1, January 1984.

Haskins, R., and Lorie, R., "On Extending the Functions of a Relational Database System," *Proceedings ACM SIGMOD Conference*, June 1982.

Lockemann, Peter C., *Multimedia Databases: A Paradigm and an Architecture*, Naval Postgraduate School, Monterey, California, August 1988.

Lum, V. Y., Wu, C. T., and Hsiao, D. K., *Integrating Advanced Techniques into Multimedia DBMS*, NPS52-87-050, Naval Postgraduate School, Monterey, CA, November 1987.

Lum, V. Y., Wu, C. T., and Hsiao, D. K., "Design of an Integrated DBMS to Support Advanced Applications," *Proceedings International Conference on the Foundations of Data Organization*, Kyoto, Japan, May 1985.

MacRecorder: Introduction To Sound, Farallon Computing, Inc., Berkeley, California, 1987.

Masunaga, Yoshifumi, "Multimedia Databases: A Formal Framework," *IEEE Computer Society Office Automation Symposium*, April 1987.

Maxemchuk, N., "An Experimental Speech Storage and Editing Facility," *Bell Systems Technical Journal*, Vol. 59, 1980.

Meyer-Wegener, K., *A Project on Multimedia Databases*, Naval Postgraduate School, Monterey, California, April 1988.

Meyer-Wegener, K., V. Lum and C. Wu, *Image Database Management in a Multimedia System*, Naval Postgraduate School, Monterey, California, April 1988.

Michel, Stephen L., "HYPERCARD: Apple's Illuminated Manuscript," *CD-ROM Review*, May 1988.

Myers, Andrew B. (ed.), "Speech Processing Technology," *AT&T Technical Journal*, Vol. 65, Issue 5, September/October 1986.

Ooi, B. C. et al., "Design Of A Multimedia File Server Using Optical Disks For Office Applications," *IEEE Computer Society Office Automation Symposium*, April 1987.

Postel, Jonathan B. et al., "An Experimental Multimedia Mail System," *ACM Transactions On Office Information Systems*, Vol. 6, No. 1, January 1988.

Sanders, Mark S. and E. McCormick, *Human Factors In Engineering and Design*, McGraw-Hill Book Company, New York, 6th ed., 1987.

Steinbrecher, David, "Optical Disks Go Head To Head With Traditional Storage Media," *Today's Office*, October 1987.

Strukhoff, Roger, "The Industry Emerges: Apple Shines In Seattle," *CD-ROM Review*, May 1988.

Strukhoff, Roger, "IRIS EYES: Intermedia Prize," *CD-ROM Review*, May 1988.

Sventek, Joseph S., "An Architecture Supporting Multi-Media Integration," *IEEE Computer Society Office Automation Symposium*, April 1987.

Tanenbaum, Andrew S., *Computer Networks*, Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1981.

Terry, Douglas B. and D. Swinhart, "Managing Stored Voice In The Etherphone System," *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988.

Thomas, R. H. et al., "DIAMOND: A Multimedia Message System Built on a Distributed Architecture," *Computer*, Vol. 18, No. 12, December 1985.

"Understanding Voice I/O Systems And Their Applications," *The American Voice Society*, April 1985.

Woelk, D., and Luther, W., *Multimedia Database Requirements*, MCC Technical Report No. DB-042-85, July 1985.

Woelk, D. and Kim, W., "Multimedia Information Management in an Object-Oriented Database System," *Proceedings of the Thirteenth VLDB Conference*, Brighton, United Kingdom, 1987.

Woelk, D., Luther, W., and Kim, W., "Multimedia Applications and Database Requirements," *Proceedings IEEE CS Office Automation Symposium*, Gaithersburg, Maryland, April 1987.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1
4.	Professor Vincent Y. Lum, Code 52Lu Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	10
5.	Klaus Meyer-Wegener Universitaet Kaiserslautern Fachbereich Informatik Postfach 30 49 6750 Kaiserslautern West Germany	1
6.	LCDR Gregory R. Sawyer Attack Squadron Forty-Two U.S. Naval Air Station Oceana Virginia Beach, Virginia 23460	2
7.	LT Cathy A. Thomas 13968 Stoney Gate Pl. San Diego, California 92128	1
8.	LT Diane M. Enbody 4693 Blue Pine Circle Lake Worth, Florida 33463	1